Chapter 4

Optimization Basics: A Machine Learning View

"If you optimize everything, you will always be unhappy."–Donald Knuth

4.1 Introduction

Many machine learning models are often cast as continuous optimization problems in multiple variables. The simplest example of such a problem is least-squares regression, which is also viewed as a fundamental problem in linear algebra. This is because solving a (consistent) system of equations is a special case of least-squares regression. In least-squares regression, one finds the *best-fit solution* to a system of equations that may or may not be consistent, and the loss corresponds to the aggregate squared error of the best fit. The special case of a consistent system of equations yields a loss value of 0. Least-squares regression has a special place in linear algebra, optimization, and machine learning, because it serves as a foundational problem in all three disciplines. Least-squares regression historically preceded the classification problem in machine learning, and the optimization models for classification were often motivated as modifications of the least-squares regression model. The main difference between least-squares regression and classification is that the predicted target variable is numerical in the former, whereas it is discrete (typically binary) in the latter. Therefore, the optimization model for linear regression needs to be "repaired" in order to make it usable for discrete target variables. This chapter will make a special effort to show how least-squares regression is so foundational to machine learning.

Most continuous optimization methods use differential calculus in one form or the other. Differential calculus is an old discipline, and it was independently invented by Isaac Newton and Gottfried Leibniz in the 17th century. The main idea of differential calculus is to provide a quantification of the *instantaneous* rate of change of an objective function with respect to each of the variables in its argument. Optimization methods based on differential calculus use the fact that the rate of change of an objective function at a particular set of values

C. C. Aggarwal, *Linear Algebra and Optimization for Machine Learning*,
https://doi.org/10.1007/978-3-030-40344-7_4

of the optimization variables provides hints on how to iteratively change the optimization variable(s) and bring them closer to an optimum solution. Such iterative algorithms are easy to implement on modern computers. Although computers had not been invented in the 17th century, Newton proposed several iterative methods to provide humans a systematic way to manually solve optimization problems (albeit with some rather tedious work). It was natural to adapt these methods later as computational algorithms, when computers were invented. This chapter will introduce the basics of optimization and the associated computational algorithms. Later chapters will expand on these ideas.

This chapter is organized as follows. The next section will discuss the basics of optimization. The notion of convexity is introduced in Section 4.3 because of its importance in machine learning. Important details of gradient descent are discussed in Section 4.4. There are several ways in which optimization problems are manifested in a different way in machine learning (than in traditional applications). This issue will be discussed in Section 4.5. Useful matrix calculus notations and identities are introduced in Section 4.6 for computing the derivatives of objective functions with respect to vectors. The least-squares regression problem is introduced in Section 4.7. The design of machine learning algorithms with discrete targets is presented in Section 4.8. Optimization models for multiway classification are discussed in Section 4.9. Coordinate descent methods are discussed in Section 4.10. A summary is given in Section 4.11.

4.2 The Basics of Optimization

An optimization problem has an *objective function* that is defined in terms of a set of variables, referred to as *optimization variables*. The goal of the optimization problem is to compute the values of the variables at which the objective function is either maximized or minimized. It is common to use a minimization form of the objective function in machine learning, and the corresponding objective function is often referred to as a *loss function*. Note that the term "loss function" often (semantically) refers to an objective function with certain types of properties quantifying a nonnegative "cost" associated with a particular configuration of variables. This term is used in the econometrics, statistics, and the machine learning communities, although the term "objective function" is a more general concept than the term "loss function." For example, a loss function is always associated with a minimization objective function, and it is often interpreted as a cost with a nonnegative value. Most objective functions in machine learning are multivariate loss functions over many variables. First, we will consider the simple case of optimization functions defined on a single variable.

4.2.1 Univariate Optimization

Consider a single-variable objective function $f(x)$ as follows:

$$f(x) = x^2 - 2x + 3 \tag{4.1}$$

This objective function is an upright parabola, which can also be expressed in the form $f(x) = (x-1)^2 + 2$. The objective function is shown in Figure 4.2(a); it clearly takes on its minimum value at $x = 1$, where the nonnegative term $(x-1)^2$ drops to 0. Note that at the minimum value, the rate of change of $f(x)$ with respect to x is zero, as the tangent to the

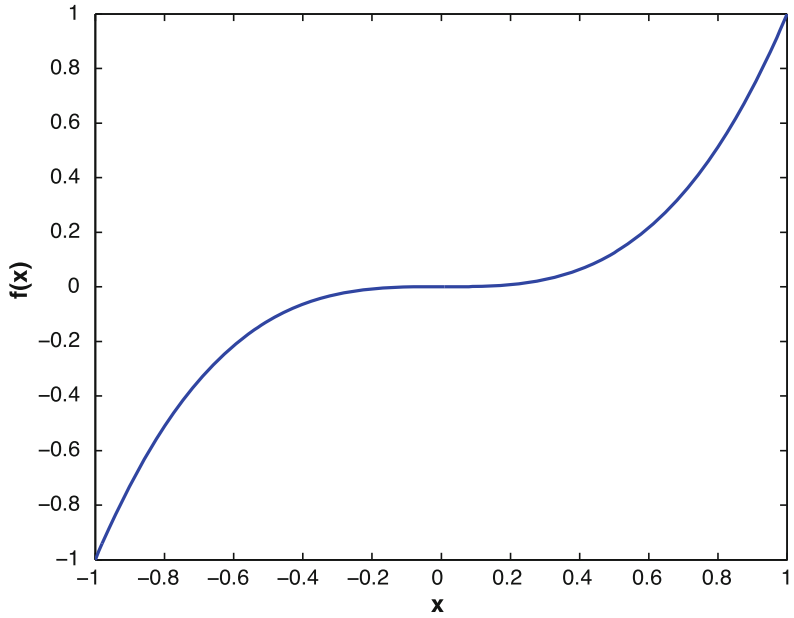


Figure 4.1: Example of 1-dimensional function $F(x) = x^3$

plot at that point is horizontal. One can also find this optimal value by computing the first derivative $f'(x)$ of the function $f(x)$ with respect to x and setting it to 0:

$$f'(x) = \frac{\mathrm{d}f(x)}{\mathrm{d}x} = 2x - 2 = 0 \tag{4.2}$$

Therefore, we obtain $x = 1$ as an optimum value. Intuitively, the function $f(x)$ changes at zero rate on slightly perturbing the value of x from $x = 1$, which suggests that it is an optimal point. However, this analysis alone is not sufficient to conclude that the point is a minimum. In order to understand this point, consider the *inverted* parabola, obtained by setting $g(x) = -f(x)$:

$$g(x) = -f(x) = -x^2 + 2x - 3 \tag{4.3}$$

Setting the derivative of $g(x)$ to 0 yields *exactly* the same solution of $x = 1$:

$$g'(x) = 2 - 2x = 0 \tag{4.4}$$

However, in this case the solution $x = 1$ is a maximum rather than a minimum. Furthermore, the point $x = 0$ is an *inflection point* or *saddle point* of the function $F(x) = x^3$ (cf. Figure 4.1), even though the derivative is 0 at $x = 0$. Such a point is neither a maximum nor a minimum.

All points for which the first derivative is zero are referred to as *critical points* of the optimization problem. A critical point might be a maximum, minimum, or saddle point. How does one distinguish between the different cases for critical points? One observation is that a function looks like an upright bowl at a minimum point, which implies that its first derivative increases at minima. In other words, the *second derivative* (i.e., derivative of the derivative) will be positive for minima (although there are a few exceptions to this rule).

For example, the second derivatives for the two quadratic functions $f(x)$ and $g(x)$ discussed above are as follows:

$$f''(x) = 2 > 0, \qquad g''(x) = -2 < 0$$

The case where the second derivative is zero is somewhat ambiguous, because such a point could be a minimum, maximum, or an inflection point. Such a critical point is referred to as *degenerate*. Therefore, for a single-variable optimization function $f(x)$ in minimization form, satisfying *both* $f'(x) = 0$ and $f''(x) > 0$ is sufficient to ensure that the point is a minimum with respect to its *immediate locality*. Such a point is referred to as a *local* minimum. This does not, however, mean that the point x is a *global* minimum across the entire range of values of x.

Lemma 4.2.1 (Optimality Conditions in Unconstrained Optimization) *A univariate function $f(x)$ is a minimum value at $x = x_0$ with respect to its immediate locality if it satisfies both $f'(x_0) = 0$ and $f''(x_0) > 0$.*

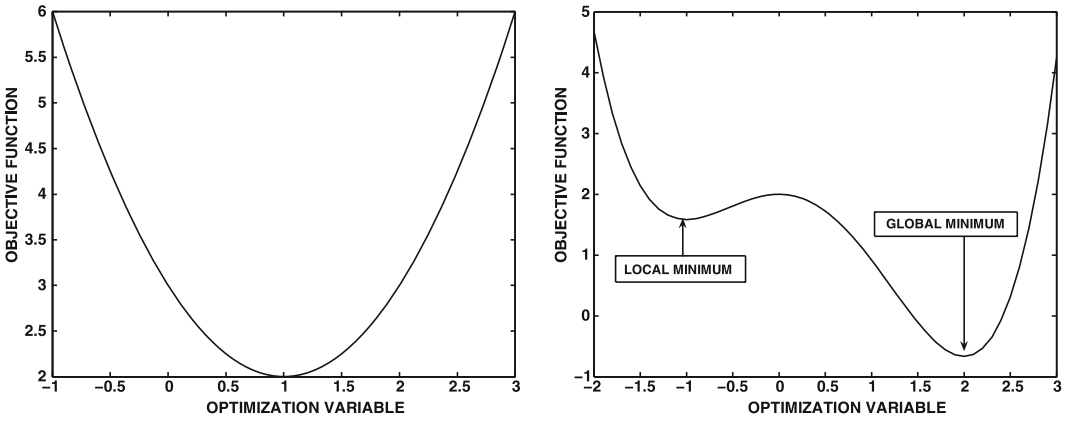
These conditions are referred to as *first-order* and *second-order* conditions for minimization. The above conditions are *sufficient* for a point to be minimum with respect to its *infinitesimal locality*, and they are "almost" *necessary* for the point to be a minimum with respect to its locality. We use the word "almost" in order to address the degenerate case where a point x_0 might satisfy $f'(x_0) = 0$ and $f''(x_0) = 0$. This type of setting is an ambiguous situation where the point x_0 might or might not be a minimum. As an example of this ambiguity, the functions $F(x) = x^3$ and $G(x) = x^4$ have zero first and second derivatives at $x = 0$, but only the latter is a minimum. One can understand the optimality condition of Lemma 4.2.1 by using a Taylor expansion of the function $f(x)$ within a small locality $x_0 + \Delta$ (cf. Section 1.5.1 of Chapter 1):

$$f(x_0 + \Delta) \approx f(x_0) + \underbrace{\Delta f'(x_0)}_{0} + \frac{\Delta^2}{2} f''(x_0)$$

Note that Δ might be either positive or negative, although Δ^2 will always be positive. The value of $|\Delta|$ is assumed to be extremely small, and successive terms rapidly drop off in magnitude. Therefore, it makes sense to keep only the first non-zero term in the above expansion in order to meaningfully compare $f(x_0)$ with $f(x_0 + \Delta)$. Since $f'(x_0)$ is zero, the first non-zero term is the second-order term containing $f''(x_0)$. Furthermore, since Δ^2 and $f''(x_0)$ are positive, it follows that $f(x_0 + \Delta) = f(x_0) + \epsilon$, where ϵ is some positive quantity. This means that $f(x_0)$ is less than $f(x_0 + \Delta)$ for any small value of Δ, whether it is positive or negative. In other words, x_0 is a minimum with respect to its immediate locality.

The Taylor expansion also provides insights as to why the degenerate case $f'(x_0) = f''(x_0) = 0$ is problematic. In the event that $f''(x)$ is zero, one would need to keep expanding the Taylor series until one reaches the first non-zero term. If the first non-zero term is positive, then one can show that $f(x_0 + \Delta) < f(x_0)$. An example of such a function is $f(x) = x^4$ at $x_0 = 0$. In such a case, x_0 is indeed a minimum with respect to its immediate locality. However, if the first non-zero term is negative or it depends on the sign of Δ, it could be a maximum or saddle point; an example is the inflection point of x^3 at the origin, which is shown in Figure 4.1.

Problem 4.2.1 *Consider the quadratic function $f(x) = ax^2 + bx + c$. Show that a point can be found at which $f(x)$ satisfies the optimality condition (for minimization) when $a > 0$. Show that the optimality condition (for maximization) is satisfied when $a < 0$.*



(a) Single global minimum
$f(x) = x^2 - 2x + 3$

(b) Global and local minimum
$F(x) = (x^4/4) - (x^3/3) - x^2 + 2$

Figure 4.2: Illustrations of local and global optima

A quadratic function is a rather simple case in which a single minimum or maximum exists, depending on the sign of the quadratic term. However, other functions have multiple turning points. For example, the function $\sin(x)$ is periodic, and has an infinite number of minima/maxima over $x \in (-\infty, +\infty)$. It is noteworthy that the optimality conditions of Lemma 4.2.1 only focus on defining a minimum in a local sense. In other words, the point is minimum with respect to its infinitesimal locality. A point that is a minimum only with respect to its immediate locality is referred to as a *local* minimum. Intuitively, the word "local" refers to the fact that the point is a minimum only within its neighborhood of (potentially) infinitesimal size. The minimum across the entire domain of values of the optimization variable is the *global* minimum. It is noteworthy that the conditions of Lemma 4.2.1 do not tell us with certainty whether or not a point is a global minimum. However, these conditions are sufficient for a point to be at least a local minimum and "almost" necessary to be a local minimum (i.e., necessary with the exception of the degenerate case discussed earlier with a zero second derivative).

Next, we will consider an objective function that has both local and global minima:

$$F(x) = (x^4/4) - (x^3/3) - x^2 + 2$$

This function is shown in Figure 4.2(b), and it has two possible minima. The minimum at $x = -1$ is a *local* minimum, and the minimum at $x = 2$ is a *global* minimum. Both the local and global minima are shown in Figure 4.2(b). On differentiating $F(x)$ with respect to x and setting it to zero, we obtain the following condition:

$$x^3 - x^2 - 2x = x(x+1)(x-2) = 0$$

The roots are $x \in \{-1, 0, 2\}$. The second derivative is $3x^2 - 2x - 2$, which is positive at -1 and 2 (minima), and negative at $x = 0$ (maximum). The value of the function at the two minima are as follows:

$$F(-1) = 1/4 + 1/3 - 1 + 2 = 19/12$$
$$F(2) = 4 - 8/3 - 4 + 2 = -2/3$$

Therefore, $x = 2$ is a *global* minimum, whereas $x = -1$ is a *local* minimum. It is noteworthy that $x = 0$ is a (local) maximum satisfying $F(0) = 2$. This local maximum appears as a small hill with a peak at $x = 0$ in Figure 4.2(b). Local optima pose a challenge for optimization problems, because there is often no way of knowing whether a solution satisfying the optimality conditions is the global optimum or not. Certain types of optimization functions, referred to as *convex* functions, are guaranteed to have a single global minimum. An example of a convex function is the univariate quadratic objective function of Figure 4.2(a). Before discussing convex functions, we will discuss the problem of reaching a solution that satisfies the conditions of Lemma 4.2.1 (and its generalization to multiple variables).

Problem 4.2.2 *Show that the function $F(x) = x^4 - 4x^3 - 2x^2 + 12x$ takes on minimum values at $x = -1$ and $x = 3$. Show that it takes on a maximum value at $x = 1$. Which of these are local optima?*

Problem 4.2.3 *Find the local and global optima of $F(x) = (x-1)^2[(x-3)^2 - 1]$. Which of these are maxima and which are minima?*

4.2.1.1 Why We Need Gradient Descent

Solving the equation $f'(x) = 0$ for x provides an *analytical* solution for a critical point. Unfortunately, it is not always possible to compute such analytical solutions in closed form. It is often difficult to exactly solve the equation $f'(x) = 0$ because this derivative might itself be a complex function of x. In other words, a *closed form solution* (like the example above) typically does not exist. For example, consider the following function that needs to be minimized:

$$f(x) = x^2 \cdot \log_e(x) - x \tag{4.5}$$

Setting the first derivative of this function to 0 yields the following condition:

$$f'(x) = 2x \cdot \log_e(x) + x - 1 = 0$$

This equation is somewhat hard to solve, although iterative methods exist for solving it. By trial and error, one might get lucky and find out that $x = 1$ is indeed a solution to the first-order optimality condition because it satisfies $f'(1) = 2\log_e(1) + 1 - 1 = 0$. Furthermore, the second derivative $f''(x)$ can be shown to be positive at $x = 1$, and therefore this point is at least a local minimum. However, solving an equation like this numerically causes all types of numerical and computational challenges; these types of challenges increase when we move from univariate optimization to multivariate optimization.

A very popular approach for optimizing objective functions (irrespective of their functional form) is to use the method of *gradient descent*. In gradient descent, one starts at an initial point $x = x_0$ and successively updates x using the *steepest descent direction*:

$$x \Leftarrow x - \alpha f'(x)$$

Here, $\alpha > 0$ regulates the step size, and is also referred to as the *learning rate*. In the univariate case, the notion of "steepest" is hard to appreciate, as there are only two directions of movement (i.e., increase x or decrease x). One of these directions causes ascent, whereas the other causes descent. However, in multivariate problems, there can be an infinite number of possible directions of descent, and the generalization of the notion of univariate derivative leads to the steepest descent direction. The value of x changes in each iteration by $\delta x = -\alpha f'(x)$. Note that at infinitesimally small values of the learning rate $\alpha > 0$, the

above updates will always reduce $f(x)$. This is because for very small α, we can use the first-order Taylor expansion to obtain the following:

$$f(x + \delta x) \approx f(x) + \delta x f'(x) = f(x) - \alpha[f'(x)]^2 < f(x) \tag{4.6}$$

Using very small values of $\alpha > 0$ is not advisable because it will take a long time for the algorithm to converge. On the other hand, using large values of α could make the effect of the update unpredictable with respect to the computed gradient (as the first-order Taylor expansion is no longer a good approximation). After all, the gradient is only an instantaneous rate of change, and it does not apply over larger ranges. Therefore, large step-sizes could cause the solution to overshoot an optimal value, if the sign of the gradient changes over the length of the step. At extremely large values of the learning rate, it is even possible for the solution to *diverge*, where it moves at an increasing speed towards large absolute values, and typically terminates with a numerical overflow.

In the following, we will show two iterations of the gradient descent procedure for the function of Equation 4.5. Consider the case where we start at $x_0 = 2$, which is larger than the optimal value of $x = 1$. At this point, the value of $f'(x)$ can be shown to be $2\log_e(2) + 1 \approx 2.4$. If we use $\alpha = 0.2$, then the value of x gets updated from x_0 as follows:

$$x_1 \Leftarrow x_0 - 0.2 * 2.4 = 2 - 0.48 = 1.52$$

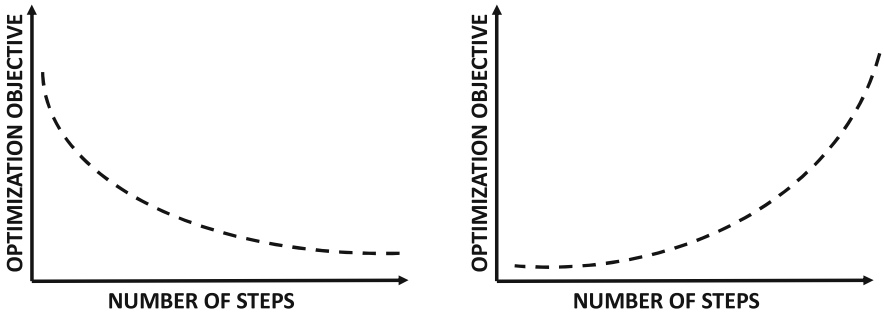
This new value of x is closer to the optimal solution. One can then recompute the derivative at $x_1 = 1.52$ and perform the update $x \Leftarrow 1.52 - 0.2 * f'(1.52)$. Performing this update again and again to construct the sequence $x_0, x_1, x_2 \ldots x_t$ will eventually converge to the optimal value of $x_t = 1$ for large values of t. Note that the choice of α does matter. For example, if we choose $\alpha = 0.8$, then it results in the following update:

$$x_1 \Leftarrow x_0 - \alpha f'(x_0) = 2 - 2.4 * 0.8 = 0.08$$

In this case, the solution has overshot the optimal value of $x = 1$, although it is still closer to the optimal solution than the initial point of $x_0 = 2$. The solution can still be shown to *converge* to an optimal value, but after a longer time. As we will see later, even this is not guaranteed in all cases.

4.2.1.2 Convergence of Gradient Descent

The execution of gradient-descent updates will generally result in a sequence of values $x_0, x_1 \ldots x_t$ of the optimization variable, which become successively closer to an optimum solution. As the value of x_t nears the optimum value, the derivative $f'(x_t)$ also tends to be closer and closer to zero (thereby satisfying the first-order optimality conditions of Lemma 4.2.1). In other words, the absolute step size will tend to reduce over the execution of the algorithm. As gradient descent nears an optimal solution, the objective function will also improve at a slower rate. This observation provides some natural ideas on making decisions regarding the termination of the algorithm (when the current solution is sufficiently close to an optimal value). The idea is to plot the current value of $f(x_t)$ with iteration index t as the algorithm progresses. A typical example of good progress during gradient descent is shown in Figure 4.3(a). The X-axis contains the iteration index, whereas the Y-axis contains the objective function value. The objective function value need not be monotonically decreasing over the course of the algorithm, but it will tend to show small noisy changes (without significant long-term direction) after some point. This situation can be treated as a good termination point for the algorithm. However, in some cases, the update steps can be shown to *diverge* from an optimal solution, if the step size is not chosen properly.



(a) Convergence (b) Divergence

Figure 4.3: Typical behaviors of objective function during convergence and divergence

4.2.1.3 The Divergence Problem

Choosing a very large learning rate α can cause overshooting from the optimal solution, and even divergence in some cases. In order to understand this point, let us consider the quadratic function $f(x)$ of Figure 4.2(a), which takes on its optimal value at $x = 1$:

$$f(x) = x^2 - 2x + 3$$

Now imagine a situation where the starting point is $x_0 = 2$, and one chooses a large learning rate $\alpha = 10$. The derivative of $f(x) = 2x - 2$ evaluates to $f'(x_0) = f'(2) = 2$. Then, the update from the first step yields the following:

$$x_1 \Leftarrow x_0 - 10 * 2 = 2 - 20 = -18$$

Note that the new point x_1 is *much further away* from the optimal value of $x = 1$, which is caused by the overshooting problem. Even worse, the absolute gradient is very large at this point, and it evaluates to $f'(-18) = -38$. If we keep the learning rate fixed, it will cause the solution to move at an even faster rate in the opposite direction:

$$x_2 \Leftarrow x_1 - 10 * (-38) = -18 + 380 = 362$$

In this case, the solution has overshot back in the original direction but is even further away from the optimal solution. Further updates cause back-and-forth movements at increasingly large amplitudes:

$$x_3 \Leftarrow x_2 - 10 * 722 = 362 - 7220 = -6858, \qquad x_4 \Leftarrow x_3 + 10 * 13718 = 130322$$

Note that each iteration flips the sign of the current solution and increases its magnitude by a factor of about 20. In other words, the solution moves away faster and faster from an optimal solution until it leads to a numerical overflow. An example of the behavior of the objective function during divergence is shown in Figure 4.3(b).

It is common to reduce the learning rate over the course of the algorithm, and one of the many purposes served by such an approach is to arrest divergence; however, in some cases, such an approach might not prevent divergence, especially if the initial learning rate is large. Therefore, when an analyst encounters a situation in gradient descent, where the size of the parameter vector seems to increase rapidly (and the optimization objective worsens),

it is a tell-tale sign of divergence. The first adjustment should be to experiment with a lower initial learning rate. However, choosing a learning rate that is too small might lead to unnecessarily slow progress, which causes the entire procedure to take too much time. There is a considerable literature in finding the correct step size or adjusting it over the course of the algorithm. Some of these issues will be discussed in later sections.

4.2.2 Bivariate Optimization

The univariate optimization scenario is rather unrealistic, and most optimization problems in real-world settings have multiple variables. In order to understand the subtle differences between single-variable and multivariable optimization, we will first consider the case of an optimization function containing two variables. This setting is referred to as *bivariate optimization*, and it is helpful in bridging the gap in complexity from single-variable optimization to multivariate optimization. For ease in understanding, we will consider bivariate generalizations of the univariate optimization functions in Figure 4.2. We construct bivariate functions by adding two instances of the univariate function shown in Figure 4.2 as follows:

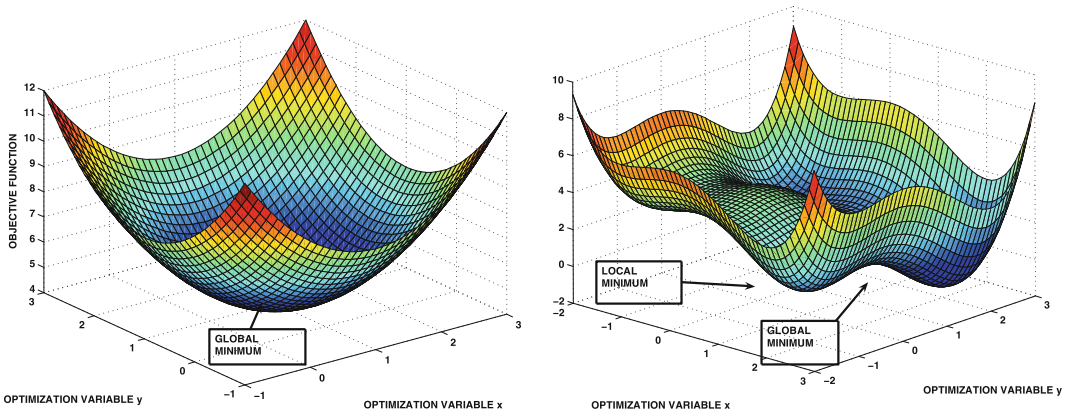
$$g(x, y) = f(x) + f(y) = x^2 + y^2 - 2x - 2y + 6$$
$$G(x, y) = g(x) + g(y) = ([x^4 + y^4]/4) - ([x^3 + y^3]/3) - x^2 - y^2 + 4$$

Note that these functions are simplified and have very special structure; they are *additively separable*. Additively separable functions are those in which univariate terms are added, and they do not interact with one another. In other words, an additively separable function might contain terms like $\sin(x^2)$ and $\sin(y^2)$, but not $\sin(xy)$. Nevertheless, these simplified polynomial functions are adequate for demonstrating the complexities associated with multivariable optimization. In fact, as discussed in Section 3.4.4 of Chapter 3, all quadratic functions can be represented in additively separable form (although this is not true for non-quadratic functions). The two bivariate functions $g(x, y)$ and $G(x, y)$ are shown in Figure 4.4(a) and (b), respectively. It is evident that the single-variable cross-sections of the objective functions in Figure 4.4(a) and (b) are similar to the 1-dimensional functions in Figure 4.2(a) and (b). The objective function of Figure 4.4(a) has a single global optimum (like the quadratic function of Figure 4.2(a) in one dimension). However, the objective function of Figure 4.4(b) has four minima, only one of which is global minimum at $[x, y] = [2, 2]$. Examples of local and global minima are annotated in Figure 4.4(b).

In this case, one can compute the *partial derivative* of the objective functions $g(x, y)$ and $G(x, y)$ (of Figure 4.2) in order to perform gradient descent. A partial derivative computes the derivative with respect to a particular variable, while treating other variables as constants. In fact, a "gradient" is naturally defined as a vector of partial derivatives. One can compute the gradient of the function $g(x, y)$ in Figure 4.4(a) as follows:

$$\nabla g(x, y) = \left[\frac{\partial g(x, y)}{\partial x}, \frac{\partial g(x, y)}{\partial y} \right]^T = \begin{bmatrix} 2x - 2 \\ 2y - 2 \end{bmatrix}$$

The notation "∇" is added in front of a function to denote its gradient. This notation will be consistently used in the book, and we will occasionally add subscripts like $\nabla_{x,y} g(x, y)$ to clarify the choice of variables with respect to which the gradient is computed. In this case, the gradient is a column vector with two components, because we have two optimization variables x and y. Each component of the 2-dimensional vector is a partial derivative of the objective function with respect to one of the two variables. The simplest approach for



(a) Single global minimum
$g(x, y) = x^2 + y^2 - 2x - 2y + 6$

(b) Global and local minimum
$G(x, y) = ([x^4 + y^4]/4)$
$-([x^3 + y^3]/3) - x^2 - y^2 + 4$

Figure 4.4: Illustrations of local and global optima

solving the optimization problem is to set the gradient $\nabla g(x, y)$ to zero, which leads to the solution $[x, y] = [1, 1]$. We will discuss the second-order optimality conditions (to distinguish between maxima, minima, and inflection points) in Section 4.2.3.

The simple approach of setting the gradient of the objective function to zero might not always lead to a system of equations with a closed-form solution. The common solution is to use gradient-descent updates with respect to the optimization variables $[x, y]$ as follows:

$$\begin{bmatrix} x_{t+1} \\ y_{t+1} \end{bmatrix} \Leftarrow \begin{bmatrix} x_t \\ y_t \end{bmatrix} - \alpha \nabla g(x_t, y_t) = \begin{bmatrix} x_t \\ y_t \end{bmatrix} - \alpha \begin{bmatrix} 2x_t - 2 \\ 2y_t - 2 \end{bmatrix}$$

So far, we have only examined additively separable functions with simple structure. Now let us consider a somewhat more complicated function:

$$H(x, y) = x^2 - \sin(xy) + y^2 - 2x$$

In such a case, the term $\sin(xy)$ ensures that the function is not additively separable. In such a case, the gradient of the function can be shown to be the following:

$$\nabla H(x, y) = \left[\frac{\partial H(x, y)}{\partial x}, \frac{\partial H(x, y)}{\partial y} \right]^T = \begin{bmatrix} 2x - y\cos(xy) - 2 \\ 2y - x\cos(xy) \end{bmatrix}$$

Although the partial derivative components are no longer expressed in terms of individual variables, gradient descent updates can be performed in a similar manner to the previous case.

As in the case of univariate optimization, the presence of local optima remains a consistent problem. For example, in the case of the function $G(x, y)$ shown in Figure 4.4(b), local optima are clearly visible. All critical points can be found by setting the gradient $\nabla G(x, y)$ to 0:

$$\nabla G(x, y) = \begin{bmatrix} x^3 - x^2 - 2x \\ y^3 - y^2 - 2y \end{bmatrix} = \overline{0}$$

This optimization problem has an interesting structure, because any of the nine pairs $(x, y) \in \{-1, 0, 2\} \times \{-1, 0, 2\}$ satisfies the first order optimality conditions, and are therefore critical points. Among these, there is a single global minimum, three local minima, and a single local maximum at $(0, 0)$. The other four can be shown to be saddle points. The classification of points as minima, maxima, or saddle points can only be accomplished with the use of multivariate second-order conditions, which are direct generalizations of the univariate optimality conditions of Lemma 4.2.1. The discussion of second-order optimality conditions for the multivariate case is deferred to Section 4.2.3. Note the rapid proliferation of the number of possible critical points satisfying the optimality conditions when the optimization problem contains two variables instead of one. In general, when a multivariate problem is posed as sum of univariate functions, the number of local optima can proliferate exponentially fast with the number of optimization variables.

Problem 4.2.4 *Consider a univariate function $f(x)$, which has k values of x satisfying the optimality condition $f'(x) = 0$. Let $G(x, y) = f(x) + f(y)$ be a bivariate objective function. Show that there are k^2 pairs (x, y) satisfying $\nabla G(x, y) = \overline{0}$. How many tuples $[x_1, \ldots, x_d]^T$ would satisfy the first-order optimality condition for the d-dimensional function $H(x_1 \ldots x_d) = \sum_{i=1}^{d} f(x_i)$?*

In the case of the objective function of Figure 4.4(b), a single (local or global) optimum exists in each of the four quadrants. Furthermore, it can be shown that starting the gradient descent in a particular quadrant (at low learning rates) will converge to the single optimum in that quadrant because each quadrant contains its own local bowl. At higher learning rates, it is possible for the gradient descent to overshoot a local/global optimum and move to a different bowl (or even behave in an unpredictable way with numerical overflows). Therefore, the final resting point of gradient descent depends on (what would seem to be) small details of the computational procedure, such as the starting point or the learning rate. We will discuss many of these details in Section 4.4.

The function $g(x, y)$ of Figure 4.4(a) has a single global optimum and no local optima. In such cases, one is more likely to reach the global optimum, irrespective of where one starts the gradient-descent procedure. The better outcome in this case is a result of the structure of the optimization problem. Many optimization problems that are encountered in machine learning have the nice structure of Figure 4.4(a) (or something very close to it), as a result of which local optima cause fewer problems than would seem at first glance.

4.2.3 Multivariate Optimization

Most machine learning problems are defined on a large parameter space containing multiple optimization variables. The variables of the optimization problem are *parameters* that are used to create a *prediction function* of either observed or hidden attributes of the machine learning problem. For example, in a linear regression problem, the optimization variables $w_1, w_2 \ldots w_d$ are used to predict the dependent variable y from the independent variables $x_1 \ldots x_d$ as follows:

$$y = \sum_{i=1}^{d} w_i x_i$$

Starting from this section, we assume that only the notations $w_1 \ldots w_d$ represent optimization variables, whereas the other "variables" like x_i and y are really observed values from the data set at hand (which are constants from the optimization perspective). This notation is typical for machine learning problems. The objective functions often penalize differences in

observed and predicted values of specific attributes, such as the variable y shown above. For example, if we have many observed tuples of the form $[x_1, x_2 \ldots x_d, y]$, one can sum up the values of $(y - \sum_{i=1}^{d} w_i x_i)^2$ over all the observed tuples. Such objective functions are often referred to as loss functions in machine learning parlance. Therefore, we will often substitute the term "objective function" with "loss function" in the remainder of this chapter. In this section, we will assume that the loss function $J(\overline{w})$ is a function of a vector of multiple optimization variables $\overline{w} = [w_1 \ldots w_d]^T$. Unlike the discussion in the preceding sections, we will use the notations $w_1 \ldots w_d$ for optimization variables, because the notations $\overline{X}$, x_i, $\overline{y}$, and y_i, will be reserved for the attributes in the data (whose values are observed). Although attributes are also sometimes referred to as "variables" (e.g., dependent and independent variables) in machine learning parlance, they are not variables from the perspective of the optimization problem. The values of the attributes are always fixed based on the observed data during training, and therefore appear among the (constant) coefficients of the optimization problem. Confusingly, these attributes (with constant observed values) are also referred to as "variables" in machine learning, because they are arguments of the prediction function that the machine learning algorithm is trying to model. The use of notations such as $\overline{X}$, x_i, $\overline{y}$, and y_i to denote attributes is a common practice in the machine learning community. Therefore, the subsequent discussion in this chapter will be consistent with this convention. The value of d corresponds to the number of optimization variables in the problem at hand, and the parameter vector $\overline{w} = [w_1 \ldots w_d]^T$ is assumed to be a column vector.

The computation of the gradient of an objective function of d variables is similar to the bivariate case discussed in the previous section. The main difference is that a d-dimensional vector of partial derivatives is computed instead of a 2-dimensional vector. The ith component of the d-dimensional gradient vector is the partial derivative of J with respect to the ith parameter w_i. The simplest approach to solve the optimization problem directly (without gradient descent) is to set the gradient vector to zero, which leads to the following set of d conditions:

$$\frac{\partial J(\overline{w})}{\partial w_i} = 0, \quad \forall i \in \{1 \ldots d\}$$

These conditions lead to a system of d equations, which can be solved to determine the parameters $w_1 \ldots w_d$. As in the case of univariate optimization, one would like to have a way to characterize whether a critical point (i.e., zero-gradient point) is a maximum, minimum, or inflection point. This brings us to the second-order condition. Recall that in single-variable optimization, the condition for $f(w)$ to be a minimum is $f''(w) > 0$. In multivariate optimization, this principle is generalized with the use of the *Hessian* matrix. Instead of a scalar second derivative, we have a $d \times d$ matrix of second-derivatives, which includes *pairwise* derivatives of J with respect to different pairs of variables. The Hessian of the loss function $J(\overline{w})$ with respect to the optimization variables $w_1 \ldots w_d$ is given by a $d \times d$ symmetric matrix H, in which the (i, j)th entry H_{ij} is defined as follows:

$$H_{ij} = \frac{\partial^2 J(\overline{w})}{\partial w_i \partial w_j} \tag{4.7}$$

Note that the (i, j)th entry of the Hessian is equal to the (j, i)th entry because partial derivatives are commutative according to *Schwarz's theorem*. The fact that the Hessian is a symmetric matrix is helpful in many computational algorithms that require eigendecomposition of the matrix.

The Hessian matrix is a direct generalization of the univariate second derivative $f''(w)$. For a univariate function, the Hessian is a 1×1 matrix containing $f''(w)$ as its only entry.

Strictly speaking, the Hessian is a *function* of $\overline{w}$, and should be denoted by $H(\overline{w})$, although we denote it by H for brevity. In the event that the function $J(\overline{w})$ is quadratic, the entries in the Hessian matrix do not depend on the parameter vector $\overline{w} = [w_1 \ldots w_d]^T$. This is similar to the univariate case, where the second derivative $f''(w)$ is a constant when the function $f(w)$ is quadratic. In general, however, the Hessian matrix depends on the value of the parameter vector $\overline{w}$ at which it is computed. For a parameter vector $\overline{w}$ at which the gradient is zero (i.e., critical point), one needs to test the Hessian matrix H in the same way we test $f''(w)$ in univariate functions. Just as $f''(w)$ needs to be positive for a point w to be a minimum, the Hessian matrix H needs to be positive-*definite* for a point to be guaranteed to be a minimum. In order to understand this point, we consider the second-order, multivariate Taylor expansion of $J(\overline{w})$ in the immediate locality of $\overline{w}_0$ along the direction $\overline{v}$ and small radius $\epsilon > 0$:

$$J(\overline{w}_0 + \epsilon\overline{v}) \approx J(\overline{w}_0) + \epsilon \overline{v}^T \underbrace{[\nabla J(\overline{w}_0)]}_{0} + \frac{\epsilon^2}{2}[\overline{v}^T H \overline{v}] \tag{4.8}$$

The Hessian matrix H, which depends on the parameter vector, is computed at $\overline{w} = \overline{w}_0$. It is evident that the objective function $J(\overline{w}_0)$ will be less than $J(\overline{w}_0 + \epsilon\overline{v})$ when we have $\overline{v}^T H \overline{v} > 0$. If we can find even a single direction $\overline{v}$ where we have $\overline{v}^T H \overline{v} < 0$, then $\overline{w}$ is clearly not a minimum with respect to its immediate locality. A matrix H that satisfies $\overline{v}^T H \overline{v} > 0$ is positive definite (cf. Section 3.3.8). The notion of positive definiteness of the Hessian is the direct generalization of the second-derivative condition $f''(w) > 0$ for univariate functions. After all, the Hessian of a univariate function is a 1×1 matrix containing the second derivative. The single entry in this matrix needs to be positive for this 1×1 matrix to be positive-definite.

Assuming that the gradient is zero at critical point $\overline{w}$, we can summarize the following second-order optimality conditions:

1. If the Hessian is positive definite at $\overline{w} = [w_1 \ldots w_d]^T$, then $\overline{w}$ is a local minimum.
2. If the Hessian is negative definite at $\overline{w} = [w_1 \ldots w_d]^T$, then $\overline{w}$ is a local maximum.
3. If the Hessian is indefinite at $\overline{w}$, then $\overline{w}$ is a saddle point.
4. If the Hessian is positive- or negative **semi**-definite, then the test is inconclusive, because the point could either be a local optimum or a saddle point.

These conditions represent direct generalizations of univariate optimality conditions. It is helpful to examine what the saddle point for an indefinite Hessian matrix looks like. Consider the following optimization objective function $g(w_1, w_2) = w_1^2 - w_2^2$. The Hessian of this quadratic function is independent of the parameter vector $[w_1, w_2]^T$, and is defined as follows:

$$H = \begin{bmatrix} 2 & 0 \\ 0 & -2 \end{bmatrix}$$

This Hessian turns out to be a diagonal matrix, which is clearly indefinite because one of the two diagonal entries is negative. The point $[0, 0]$ is a critical point because the gradient is zero at that point. However, this point is a saddle point because of the indefinite nature of the Hessian matrix. This saddle point is illustrated in Figure 4.5.

Problem 4.2.5 *The gradient of the objective function $J(\overline{w})$ is 0 and the determinant of the Hessian is negative at $\overline{w} = \overline{w}_0$. Is $\overline{w}_0$ a minimum, maximum, or a saddle-point?*

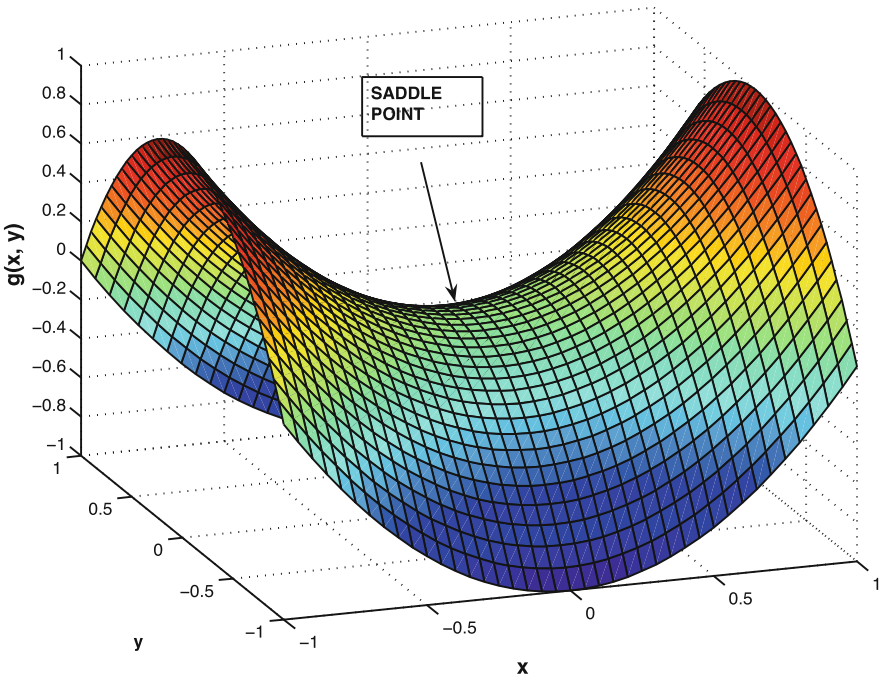


Figure 4.5: Re-visiting Figure 3.6: Illustration of saddle point created by indefinite Hessian

Setting the gradient of the objective function to 0 and then solving the resulting system of equations is usually computationally difficult. Therefore, gradient-descent is used. In other words, we use the following updates repeatedly with learning rate α:

$$[w_1 \ldots w_d]^T \Leftarrow [w_1 \ldots w_d]^T - \alpha \left[\frac{\partial J(\overline{w})}{\partial w_1} \ldots \frac{\partial J(\overline{w})}{\partial w_d} \right]^T \tag{4.9}$$

One can also write the above expression in terms of the gradient of the objective function with respect to $\overline{w}$:

$$\overline{w} \Leftarrow \overline{w} - \alpha \nabla J(\overline{w})$$

Here, $\nabla J(\overline{w})$ is a column vector containing the partial derivatives of $J(\overline{w})$ with respect to the different parameters in the column vector $\overline{w}$. Although the learning rate α is shown as a constant here, it usually varies over the course of the algorithm (cf. Section 4.4.2).

4.3 Convex Objective Functions

The presence of local minima creates uncertainty about the effectiveness of gradient-descent algorithms. Ideally, one would like to have an objective function without local minima. A specific type of objective function with this property is the class of *convex* functions. First, we need to define the concept of *convex sets*, as convex functions are defined only with domains that are convex.

Definition 4.3.1 (Convex Set) *A set S is convex, if for every pair of points $\overline{w}_1, \overline{w}_2 \in S$, the point $\lambda \overline{w}_1 + [1 - \lambda]\overline{w}_2$ must also be in S for all $\lambda \in (0, 1)$.*

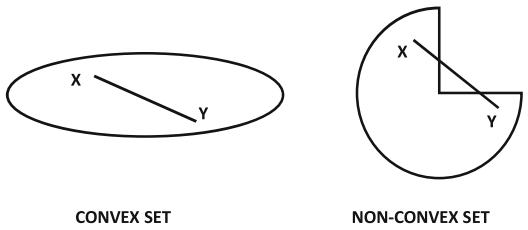


Figure 4.6: Examples of convex and non-convex sets

In other words, it is impossible to find a pair of points in the set such that any of the points on the straight line joining them do not lie in the set. A *closed convex set* is one in which the boundary points (i.e., limit points) of the set are included within the set, whereas an *open convex set* is one in which all points within the boundary are included but not the boundary itself. For example, in 1-dimensional space the set is $[-2, +2]$ is a closed convex set, whereas the set $(-2, +2)$ is an open convex set.

Examples of convex and non-convex sets are illustrated in Figure 4.6. A circle, an ellipse, a square, or a half-moon are all convex sets. However, a three-quarter circle is not a convex set because one can draw a line between the two points inside the set, so that a portion of the line lies outside the set (cf. Figure 4.6).

A convex function $F(\overline{w})$ is defined as a function with a convex domain that satisfies the following condition for any $\lambda \in (0, 1)$:

$$F(\lambda \overline{w}_1 + (1 - \lambda)\overline{w}_2) \leq \lambda F(\overline{w}_1) + (1 - \lambda)F(\overline{w}_2) \tag{4.10}$$

One can generalize the convexity condition to k points, as discussed in the practice problem below.

Problem 4.3.1 *For a convex function $F(\cdot)$, and k parameter vectors $\overline{w}_1 \ldots \overline{w}_k$, show that the following is true for any $\lambda_1 \ldots \lambda_k \geq 0$ and satisfying $\sum_i \lambda_i = 1$:*

$$F(\sum_{i=1}^{k} \lambda_i \overline{w}_i) \leq \sum_{i=1}^{k} \lambda_i F(\overline{w}_i)$$

The simplest example of a convex objective function is the class of quadratic functions in which the leading (quadratic) term has a nonnegative coefficient:

$$f(w) = a \cdot w^2 + b \cdot w + c$$

Here, a needs to be nonnegative for the function to be considered quadratic. The result can be shown by using the convexity condition above. All linear functions are always convex, because the convexity property holds with equality.

Lemma 4.3.1 *A linear function of the vector $\overline{w}$ is always convex.*

Convex functions have a number of useful properties that are leveraged in practical applications.

Lemma 4.3.2 *Convex functions obey the following properties:*

1. *The sum of convex functions is always convex.*
2. *The maximum of convex functions is convex.*
3. *The square of a nonnegative convex function is convex.*

4. *If $F(\cdot)$ is a convex function with a single argument and $G(\overline{w})$ is a linear function with a scalar output, then $F(G(\overline{w}))$ is convex.*

5. *If $F(\cdot)$ is a convex non-increasing function and $G(\overline{w})$ is a concave function with a scalar output, then $F(G(\overline{w}))$ is convex.*

6. *If $F(\cdot)$ is a convex non-decreasing function and $G(\overline{w})$ is a convex function with a scalar output, then $F(G(\overline{w}))$ is convex.*

We leave the detailed proofs of these results (which can be derived from Equation 4.10) as an exercise:

Problem 4.3.2 *Prove all the results of Lemma 4.3.2 using the definition of convexity.*

There are several natural combinations of convex functions that one might expect to be convex at first glance, but turn out to be non-convex on closer examination. The product of two convex functions is not necessarily convex. The functions $f(x) = x$ and $g(x) = x^2$ are convex functions, but their product $h(x) = f(x) \cdot g(x) = x^3$ is not convex (see Figure 4.1). Furthermore, the composition of two convex functions is not necessarily convex, and it might be indefinite or concave. As a specific example, consider the linear convex function $f(x) = -x$ and also the quadratic convex function $g(x) = x^2$. Then, we have $f(g(x)) = -x^2$, which is a concave function. The result on the composition of functions is important from the perspective of deep neural networks (cf. Chapter 11). *Even though the individual nodes of neural networks usually compute convex functions, the composition of the functions computed by successive nodes is often not convex.*

A nice property of convex functions is that a local minimum will also be a global minimum. If there are two "local" minima, then the above convexity condition ensures that the entire line joining them also has the same objective function value.

Problem 4.3.3 *Use the convexity condition to show that every local minimum in a convex function must also be a global minimum.*

The fact that every local minimum is a global minimum can also be characterized by using a geometric definition of convexity. This geometric definition, which is also referred to as the *first-derivative condition*, is that the entire convex function will always lie above a tangent to a convex function, as shown in Figure 4.7. This figure illustrates a 2-dimensional convex function, where the horizontal directions are arguments to the function (i.e., optimization variables), and the vertical direction is the objective function value. An important consequence of convexity is that one is often guaranteed to reach a global optimum if successful convergence occurs during the gradient-descent procedure.

The condition of Figure 4.7 can also be written algebraically using the gradient of the convex function at a given point $\overline{w}_0$. In fact, this condition provides an alternative definition of convexity. We summarize this condition below:

Lemma 4.3.3 (First-Derivative Characterization of Convexity) *A differentiable function $F(\overline{w})$ is a convex function if and only if the following is true for any pair $\overline{w}_0$ and $\overline{w}$:*

$$F(\overline{w}) \geq F(\overline{w}_0) + [\nabla F(\overline{w}_0)] \cdot (\overline{w} - \overline{w}_0)$$

We omit a detailed proof of the lemma. Note that if the gradient of $F(\overline{w})$ is zero at $\overline{w} = \overline{w}_0$, it would imply that $F(\overline{w}) \geq F(\overline{w}_0)$ for any $\overline{w}$. In other words, $\overline{w}_0$ is a global minimum. Therefore, any critical point that satisfies the first-derivative condition is a global minimum.

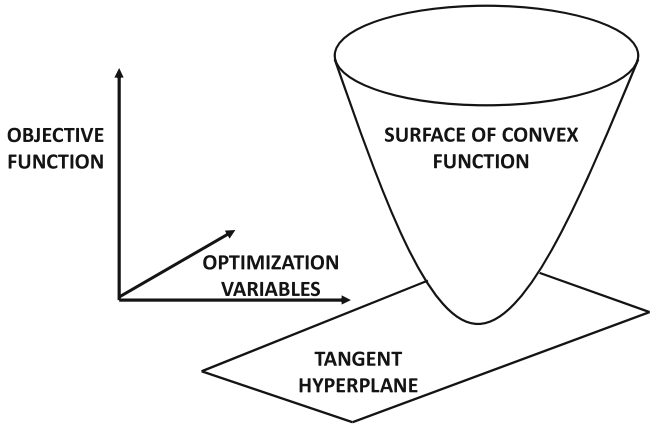


Figure 4.7: A convex function always lies entirely above any tangent to the surface. The example illustrates a 2-dimensional function, where the two horizontal axes are the optimization variables and the vertical axis is the objective function value

The main disadvantage of the first-derivative condition (with respect to the direct definition of convexity) is that it applies only to differentiable functions. Interestingly, there is a third characterization of convexity in terms of the second-derivative:

Lemma 4.3.4 (Second-Derivative Characterization of Convexity) *The twice differentiable function $F(\overline{w})$ is convex, if and only if it has a positive semidefinite Hessian at every value of the parameter $\overline{w}$ in the domain of $F(\cdot)$.*

The second derivative condition has the disadvantage of requiring the function $F(\overline{w})$ to be twice differentiable. Therefore, the following convexity definitions are equivalent for twice-differentiable functions defined over $\mathcal{R}^d$:

1. **Direct:** The convexity condition $F(\lambda\overline{w}_1 + [1-\lambda]\overline{w}_2) \leq \lambda F(\overline{w}_1) + (1-\lambda)F(\overline{w}_2)$ is satisfied for all $\overline{w}_1, \overline{w}_2$ and $\lambda \in (0,1)$.

2. **First-derivative:** The first-derivative condition $F(\overline{w}) \geq F(\overline{w}_0) + [\nabla F(\overline{w}_0)] \cdot (\overline{w} - \overline{w}_0)$ is satisfied for all $\overline{w}$ and $\overline{w}_0$.

3. **Second-derivative:** The Hessian of $F(\overline{w})$ is positive semidefinite for all $\overline{w}$.

One can choose to use any of the above conditions as the definition of convexity, and then derive the other two as lemmas. However, the direct definition is slightly more general because it does not depend on differentiability, whereas the other definitions have the additional requirement of differentiability. For example, the function $F(\overline{w}) = \|\overline{w}\|_1$ is convex but only the first definition can be used because of its non-differentiability at any point where a component of $\overline{w}$ is 0. We refer the reader to [10, 15, 22] for detailed proofs of the equivalence of the various definitions in the differentiable case. It is often the case that a particular definition is easier to use than another when one is trying to prove the convexity of a specific function. Many machine learning objective functions are of the form $F(G(\overline{w}))$, where $G(\overline{w})$ is the linear function $\overline{w} \cdot \overline{X}^T$ for a row vector containing a d-dimensional data point $\overline{X}$, and $F(\cdot)$ is a univariate function. In such a case, one only needs to prove that the univariate function $F(\cdot)$ is convex, based on the final portion of Lemma 4.3.2. It is

particularly easy to use the second-order condition $F''(\cdot) \geq 0$ for univariate functions. As a specific example, we provide a practice exercise for showing the convexity of the logarithmic logistic loss function. This function is useful for showing the convexity of logistic regression.

Problem 4.3.4 *Use the second derivative condition to show that the univariate function $F(x) = log_e(1 + exp(-x))$ is convex.*

Problem 4.3.5 *Use the second-derivative condition to show that if the univariate function $F(x)$ is convex, then the function $G(x) = F(-x)$ must be convex as well.*

A slightly stronger condition than convexity is *strict convexity* in which the convexity condition is modified to strict inequality. A strictly convex function $F(\overline{w})$ is defined as a function that satisfies the following condition for any $\lambda \in (0, 1)$:

$$F(\lambda \overline{w}_1 + (1 - \lambda)\overline{w}_2) < \lambda F(\overline{w}_1) + (1 - \lambda)F(\overline{w}_2)$$

For example, a bowl with a flat bottom is convex, but it is not strictly convex. A strictly convex function will have a unique global minimum. One can also adapt the first-order conditions to strictly convex functions. A function $F(\cdot)$ can be shown to be strictly convex if and only if the following condition holds for all $\overline{w}$ and $\overline{w}_0$:

$$F(\overline{w}) > F(\overline{w}_0) + [\nabla F(\overline{w}_0)] \cdot (\overline{w} - \overline{w}_0)$$

The second-derivative condition cannot, however, be fully generalized to strict convexity. If a function has a positive definite Hessian everywhere, then it is guaranteed to be strictly convex. However, the converse does not necessarily hold. For example, the function $f(x) = x^4$ is strictly convex, but its second derivative is 0 at $x = 0$. An important property of strictly convex functions is the following:

Lemma 4.3.5 *A strictly convex function can contain at most one critical point. If such a point exists, it will be the global minimum of the strictly convex function.*

The above property is easy to show by using either the direct definition or the first-order definition of strict convexity. One often constructs objective functions in machine learning by adding convex and strictly convex functions. In such cases, the sum of these functions is strictly convex.

Lemma 4.3.6 *The sum of a convex function and a strictly convex function is strictly convex.*

The proof of this lemma is not very different from that of the proof of Lemma 4.3.2 for the sum of two convex functions. Many objective functions in machine learning are convex, and they can often be made strictly convex by adding a strictly convex regularizer.

A special case of convex functions is that of quadratic convex functions, which can be directly expressed in terms of the positive semidefinite Hessian. Although the Hessian of a function depends on the value of the parameter vector at a specific point, it is a constant matrix in the case of quadratic functions. An example of a quadratic convex function $f(\overline{w})$ in terms of the constant Hessian matrix H is the following:

$$f(\overline{w}) = \frac{1}{2}[\overline{w} - \overline{b}]^T H[\overline{w} - \overline{b}] + c$$

Here, $\overline{b}$ is a d-dimensional column vector, and c is a scalar. The properties of such convex functions are discussed in Chapter 3. A convex objective function is an ideal setting for a

gradient-descent algorithm; the approach will never get stuck in a local minimum. Although the objective functions in complex machine learning models (like neural networks) are not convex, they are often close to convex. As a result, gradient-descent methods work quite well in spite of the presence of local optima.

For any convex function $F(\overline{w})$, the region of the space bounded by $F(\overline{w}) \leq b$ for any constant b can be shown to be a convex set. This type of constraint is encountered often in optimization problems. Such problems are easier to solve because of the convexity of the space in which one wants to search for the parameter vector.

4.4 The Minutiae of Gradient Descent

An earlier section introduces gradient descent, which serves as the workhorse of much of optimization in machine learning. However, as the example in Section 4.2.1.3 shows, small details do matter; an improper choice of the learning rate can cause divergence of gradient descent, rather than convergence. This section discusses these important minutiae.

4.4.1 Checking Gradient Correctness with Finite Differences

Many machine learning algorithms use complex objective functions over millions of parameters. The gradients are computed either analytically and then hand-coded into the algorithm, or they are computed using automatic differentiation methods in applications like neural networks (cf. Chapter 11). In all these cases, analytical or coding errors remain a real possibility, which may or may not become obvious during execution. Knowing the reason for the poor performance of an algorithm is a critical step in deciding whether to simply debug the algorithm or to make fundamental design changes.

Consider a situation where we compute the gradient of the objective function $J(\overline{w}) = J(w_1 \ldots w_d)$. In the finite-difference method, we sample a few of the optimization parameters from $w_1 \ldots w_d$ and check their partial derivatives using the *finite-difference approximation*. The basic idea is to perturb an optimization parameter w_i by a small amount Δ and approximate the partial derivative with respect to w_i by using the difference between the perturbed value of the objective function and the original value:

$$\frac{\partial J(\overline{w})}{\partial w_i} \approx \frac{J(w_1 \ldots, w_i + \Delta, \ldots, w_d) - J(w_1, \ldots, w_i, \ldots, w_d)}{\Delta}$$

This way of estimating the gradient is referred to as a finite-difference approximation. As the name suggests, one would not obtain an exact value of the partial derivative in this way. However, in cases where the gradients are computed incorrectly, the value of the finite-difference approximation is often so wildly different from the analytical value that the error becomes self-evident. Typically, it suffices to check the partial derivatives of a small subset of the parameters in order to detect a systemic problem in gradient computation.

4.4.2 Learning Rate Decay and Bold Driver

A constant learning rate often poses a dilemma to the analyst. A lower learning rate used early on will cause the algorithm to take too long to reach anywhere close to an optimal solution. On the other hand, a large initial learning rate will allow the algorithm to come reasonably close to a good solution at first; however, the algorithm will then oscillate around the point for a very long time. Allowing the learning rate to decay over time can naturally

achieve the desired learning-rate adjustment to avoid these challenges. Therefore, a decaying learning rate α_t is subscripted with the time-stamp t, and the update is as follows:

$$\overline{w} \Leftarrow \overline{w} - \alpha_t \nabla J$$

The time t is typically measured in terms of the number of cycles over all training points. The two most common decay functions are *exponential decay* and *inverse decay*. The learning rate α_t can be expressed in terms of the initial decay rate α_0 and time t as follows:

$$\alpha_t = \alpha_0 \exp(-k \cdot t) \quad \text{[Exponential Decay]}$$
$$\alpha_t = \frac{\alpha_0}{1 + k \cdot t} \quad \text{[Inverse Decay]}$$

The parameter k controls the rate of the decay. Another approach is to use step decay in which the learning rate is reduced by a particular factor every few steps of gradient descent.

Another popular approach for adjusting the learning rate is the *bold-driver algorithm*. In the bold-driver algorithm, the learning rate changes, depending on whether the objective function is improving or worsening. The learning rate is *increased* by factor of around 5% in each iteration as long as the steps improve the objective function. As soon as the objective function *worsens* because of a step, the step is *undone* and an attempt is made again with the learning rate reduced by a factor of around 50%. This process is continued to convergence. A tricky aspect of the bold-driver algorithm is that it does not work in some noisy settings of gradient descent, where the objective function is approximated by using samples of the data. An example of such a noisy setting is *stochastic gradient descent*, which is discussed later in this chapter. In such cases, it is important to test the objective function and adjust the learning rate after m steps, rather than a single step. The change in objective function can be measured more robustly across multiple steps, and all m steps must be undone when the objective function worsens over these steps.

4.4.3 Line Search

Line search directly uses the optimum step size in order to provide the best improvement. Although it is rarely used in vanilla gradient descent (because it is computationally expensive), it is helpful in some specialized variations of gradient descent. Some inexact variations (like the *Armijo rule*) can be used in vanilla gradient descent because of their efficiency.

Let $J(\overline{w})$ be the objective function being optimized and $\overline{g}_t$ be the descent direction at the beginning of the tth step with parameter vector $\overline{w}_t$. In the steepest-descent method, the direction $\overline{g}_t$ is the same as $-\nabla J(\overline{w}_t)$, although advanced methods (see next chapter) might use other descent directions. In the following, we will not assume that $\overline{g}_t$ is the steepest-descent direction in order to preserve generality of the exposition. Clearly, the parameter vector needs to be updated as follows:

$$\overline{w}_{t+1} \Leftarrow \overline{w}_t + \alpha_t \overline{g}_t$$

In line search, the learning rate α_t is chosen in each step, so as to minimize the value of the objective function at $\overline{w}_{t+1}$. The step-size α_t is computed as follows:

$$\alpha_t = \text{argmin}_\alpha J(\overline{w}_t + \alpha \overline{g}_t) \tag{4.11}$$

After performing the step, the gradient is computed at $\overline{w}_{t+1}$ for the next step. The gradient at $\overline{w}_{t+1}$ will be perpendicular to the search direction $\overline{g}_t$ or else α_t will not be optimal. This

result can be shown by observing that if the gradient of the objective function at $\overline{w}_t + \alpha_t \overline{g}_t$ has a non-zero dot product with the current movement direction $\overline{g}_t$, then one can improve the objective function by moving an amount of either $+\delta$ or $-\delta$ along $\overline{g}_t$ from $\overline{w}_{t+1}$:

$$J(\overline{w}_t + \alpha_t \overline{g}_t \pm \delta\, \overline{g}_t) \approx J(\overline{w}_t + \alpha_t \overline{g}_t) \pm \delta \underbrace{\overline{g}_t^T [\nabla J(\overline{w}_t + \alpha_t \overline{g}_t)]}_{0} \quad \text{[Taylor Expansion]}$$

Therefore, we obtain the following:

$$\overline{g}_t^T [\nabla J(\overline{w}_t + \alpha_t \overline{g}_t)] = 0$$

We summarize the result below:

Lemma 4.4.1 *The gradient at the optimal point of a line search is always orthogonal to the current search direction.*

A natural question arises as to how the minimization of Equation 4.11 is performed. One important property of typical line-search settings is that the objective function $H(\alpha) = J(\overline{w}_t + \alpha \overline{g}_t)$, when expressed in terms of α is often a unimodal function. The main reason for this is that typical machine learning settings that use the line-search method use quadratic, convex approximations of the original objective function on which the search is done. Examples of such techniques include the *Newton method* and the *conjugate gradient method* (cf. Chapter 5).

The first step is to identify a range $[0, \alpha_{max}]$ in which to perform the search. This can be performed efficiently by evaluating the objective function value at geometrically increasing values of α (increasing every time by a factor of 2). Subsequently, it is possible to use a variety of methods to narrow the interval such as the *binary-search method*, the *golden-section search method*, and the *Armijo rule*. The first two of these methods and exact methods, and they leverage the unimodality of the objective function in terms of the step-size α. The Armijo rule is inexact, and it works even when $H(\alpha) = J(\overline{w}_t + \alpha \overline{g}_t)$ is multimodal/nonconvex in α. Therefore, the Armijo rule has broader use than exact line-search methods, especially as far as simple forms of gradient descent are concerned. In the following, we discuss these different methods.

4.4.3.1 Binary Search

We start by initializing the binary search interval for α to $[a, b] = [0, \alpha_{max}]$. In binary search over $[a, b]$, the interval is narrowed by evaluating the objective function at two closely spaced points near $(a + b)/2$. We evaluate the objective function at $(a + b)/2$ and $(a + b)/2 + \epsilon$, where ϵ is a numerically small value like 10^{-6}. In other words, we compute $H[(a+b)/2]$ and $H[(a+b)/2+\epsilon]$. This allows us to evaluate whether the function is increasing or decreasing at $(a+b)/2$ by determining which of the two evaluations is larger. If the function is increasing at $(a + b)/2$, the interval is narrowed to $[a, (a + b)/2 + \epsilon]$. Otherwise, it is narrowed to $[(a + b)/2, b]$. This process is repeated, until an interval is reached with the required level of accuracy.

4.4.3.2 Golden-Section Search

As in the case of binary search, we start by initializing $[a, b] = [0, \alpha_{max}]$. However, the process of narrowing the interval is different. The basic principle in golden-section search is to use the fact that if we pick any pair of middle samples m_1, m_2 for α in the interval $[a, b]$,

where $a < m_1 < m_2 < b$, at least one of the intervals $[a, m_1]$ and $[m_2, b]$ can be dropped. In some cases, an even larger interval like $[a, m_2]$ and $[m_1, b]$ can be dropped. This is because the minimum value for a unimodal function must always lie in an adjacent interval to the choice of $\alpha \in \{a, m_1, m_2, b\}$ that yields the minimum value of $H(\alpha)$. When $\alpha = a$ yields the minimum value for $H(\alpha)$, we can exclude the interval $(m_1, b]$, and when $\alpha = b$ yields the minimum value for $H(\alpha)$, we can exclude the interval $[a, m_2)$. When $\alpha = m_1$ yields the minimum value, we can exclude the interval $(m_2, b]$, and when $\alpha = m_2$ yields the minimum value, we can exclude the interval $[a, m_1)$. The new bounds $[a, b]$ for the interval are reset based on these exclusions. At the end of the process, we are left with an interval containing either 0 or 1 evaluated point. If we have an interval containing no evaluated point, we first select a random point $\alpha = p$ in the (reset) interval $[a, b]$, and then another random point $\alpha = q$ in the larger of the two intervals $[a, p]$ and $[p, b]$. On the other hand, if we are left with an interval $[a, b]$ containing a single evaluated point $\alpha = p$, then we select $\alpha = q$ in the larger of the two intervals $[a, p]$ and $[p, b]$. This yields another set of four points over which we can apply golden-section search. This process is repeated until an interval is reached with the required level of accuracy.

4.4.3.3 Armijo Rule

The basic idea behind the Armijo rule is that the descent direction $\overline{g}_t$ at the starting point $\overline{w}_t$ (i.e., at $\alpha = 0$) often deteriorates in terms of rate of improvement of objective function as one moves further along this direction. The rate of improvement of the objective function along the search direction at the starting point is $|\overline{g}_t^T[\nabla F(\overline{w}_t)]|$. Therefore, the (typical) improvement of the objective function at a particular value of α can optimistically be expected to be $\alpha|\overline{g}_t^T[\nabla F(\overline{w}_t)]|$ for most[1] real-world objective functions. The Armijo rule is satisfied with a fraction $\mu \in (0, 0.5)$ of this improvement. A typical value of μ is around 0.25. In other words, we want to find the largest step-size α satisfying the following:

$$F(\overline{w}_t) - F(\overline{w}_t + \alpha \overline{g}_t) \geq \mu\alpha|\overline{g}_t^T[\nabla F(\overline{w}_t)]|$$

Note that for small enough values of α, the condition above will always be satisfied. In fact, one can show using the finite-difference approximation, that for infinitesimally small values of α, the condition above is satisfied at $\mu = 1$. However, we want a larger step size to ensure faster progress. What is the largest step-size one can use? We test successively *decreasing* values of α for the condition above, and stop the first time the condition above is satisfied. In backtracking line search, we start by testing $H(\alpha_{max})$, $H(\beta\alpha_{max}) \ldots H(\beta^r\alpha_{max})$, until the condition above is satisfied. At that point we use $\alpha = \beta^r\alpha_{max}$. Here, β is a parameter drawn from $(0, 1)$, and a typical value is 0.5.

When to Use Line Search

Although the line-search method can be shown to converge to at least a local optimum, it is expensive. This is the reason that it is rarely used in vanilla gradient descent. However, it is used in some specialized variations of gradient descent like *Newton's method* (cf. Section 5.4 of Chapter 5). Exact line search is required in some of these variations, whereas fast, inexact methods like the Armijo rule can be used in vanilla gradient descent. When exact line search is required, the number of steps is often relatively small, and the fewer number of steps more

[1] It is possible to construct pathological counter-examples where this is not true.

than compensate for the expensive nature of the individual steps. An important point with the use of line-search is that convergence is guaranteed, even if the resulting solution is a local optimum.

4.4.4 Initialization

The gradient-descent procedure always starts at an initial point, and successively improves the parameter vector at a particular learning rate. A critical question arises as to how the initialization point can be chosen. For some of the relatively simple problems in machine learning (like the ones discussed in this chapter), the vector components of the initialization point can be chosen as small random values from $[-1, +1]$. In case the parameters are constrained to be nonnegative, the vector components can be chosen from $[0, 1]$.

However, this simple way of initialization can sometimes cause problems for more complex algorithms. For example, in the case of neural networks, the parameters have complex dependencies on one another, and choosing good initialization points can be critical. In other cases, choosing improper magnitudes of the initial parameters can cause numerical overflows or underflows during the updates. It is sometimes effective to use some form of heuristic optimization for initialization. Such an approach already *pretrains* the algorithm to an initialization near an optimum point. The choice of the heuristic generally depends on the algorithm at hand. Some learning algorithms like neural networks have systematic ways of performing pretraining and choosing good initializations. In this chapter, we will give some examples of heuristic initializations.

4.5 Properties of Optimization in Machine Learning

The optimization problems in machine learning have some typical properties that are often not encountered in other generic optimization settings. This section will provide an overview of these specific quirks of optimization in machine learning.

4.5.1 Typical Objective Functions and Additive Separability

Most objective functions in machine learning penalize the deviation of a *predicted value* from an *observed value* in one form or another. For example, the objective function of least-squares regression is as follows:

$$J(\overline{w}) = \sum_{i=1}^{n} \|\overline{w} \cdot \overline{X}_i^T - y_i\|^2 \tag{4.12}$$

Here, $\overline{X}_i$ is a d-dimensional row vector containing the ith of n training points, $\overline{w}$ is a d-dimensional column vector of optimization variables, and y_i contains the real-valued observation of the ith training point. Note that this objective function represents an additively separable sum of squared differences between the *predicted values* $\hat{y}_i = \overline{w} \cdot \overline{X}_i^T$ and the *observed values* y_i in the actual data.

Another form of penalization is the negative *log-likelihood objective function*. This form of the objective function uses the probability that the model's prediction of a dependent variable matches the observed value in the data. Clearly, higher values of the probability are desirable, and therefore the model should learn parameters that maximize these probabilities (or *likelihoods*). For example, such a model might output the probability of each class in a binary classification setting, and it is desired to maximize the probability of the true

(observed) class. For the ith training point, this probability is denoted by $P(\overline{X}_i, y_i, \overline{w})$, which depends on the parameter vector $\overline{w}$ and training pair $(\overline{X}_i, y_i)$. The probability of correct prediction over all training points is given by the products of probabilities $P(\overline{X}_i, y_i, \overline{w})$ over all $(\overline{X}_i, y_i)$. The negative logarithm is applied to this product to convert the maximization problem into a minimization problem (while addressing numerical underflow issues caused by repeated multiplication):

$$J(\overline{w}) = -\log_e \left[\prod_{i=1}^{n} P(\overline{X}_i, y_i, \overline{w}) \right] = -\sum_{i=1}^{n} \log_e \left[P(\overline{X}_i, y_i, \overline{w}) \right] \tag{4.13}$$

Using the logarithm also makes the objective function appear as an *additively separable sum* over the training points.

As evident from the aforementioned examples, many machine learning problems use additively separable data-centric objective functions, whether squared loss or log-likelihood loss is used. This means that each individual data point creates a small (additive) component of the objective function. In each case, the objective function contains n additively separable terms, and each point-specific error [such as $J_i = (y_i - \overline{w} \cdot \overline{X}_i^T)^2$ in least-squares regression] can be viewed as a point-specific loss. Therefore, the overall objective function can be expressed as the sum of these point-specific losses:

$$J(\overline{w}) = \sum_{i=1}^{n} J_i(\overline{w}) \tag{4.14}$$

This type of linear separability is useful, because it enables the use of fast optimization methods like *stochastic gradient descent* and *mini-batch stochastic gradient descent*, where one can replace the objective function with a sampled approximation.

4.5.2 Stochastic Gradient Descent

The linear and additive nature of the objective functions in machine learning, enables the use of techniques referred to as *stochastic gradient descent*. Stochastic gradient descent is particularly useful in the case in which the data sets are very large and one can often estimate good descent directions using modest samples of the data. Consider a sample S of the n data points $\overline{X}_1 \ldots \overline{X}_n$, where S contains the indices of the relevant data points from $\{1 \ldots n\}$. The set S of data points is referred to as a *mini-batch*. One can set up a sample-centric objective function $J(S)$ as follows:

$$J(S) = \frac{1}{2} \sum_{i \in S} (y_i - \overline{w} \cdot \overline{X}_i^T)^2 \tag{4.15}$$

The key idea in mini-batch stochastic gradient descent is that *the gradient of $J(S)$ with respect to the parameter vector $\overline{w}$ is an excellent approximation of the gradient of the full objective function J*. Therefore, the gradient-descent update of Equation 4.9 is modified to mini-batch stochastic gradient descent as follows:

$$[w_1 \ldots w_d]^T \Leftarrow [w_1 \ldots w_d]^T - \alpha \left[\frac{\partial J(S)}{\partial w_1} \ldots \frac{\partial J(S)}{\partial w_d} \right]^T \tag{4.16}$$

This approach is referred to as *mini-batch* stochastic gradient descent. Note that computing the gradient of $J(S)$ is far less computationally intensive compared to computing the gradient of the full objective function. A special case of mini-batch stochastic gradient descent is one in which the set S contains a single randomly chosen data point. This approach is referred to as stochastic gradient descent. The use of stochastic gradient descent is rare, and

one tends to use the mini-batch method more often. Typical mini-batch sizes are powers of 2, such as 64, 128, 256, and so on. The reason for this is purely practical rather than mathematical; using powers of 2 for mini-batch sizes often results in the most efficient use of resources such as Graphics Processor Units (GPUs).

Stochastic gradient-descent methods typically cycle through the full data set, rather than simply sampling the data points at random. In other words, the data points are permuted in some random order and blocks of points are drawn from this ordering. Therefore, all other points are processed before arriving at a data point again. Each cycle of the mini-batch stochastic gradient descent procedure is referred to as an *epoch*. In the case where the mini-batch size is 1, an epoch will contain n updates, where n is the training data size. In the case where the mini-batch size is k, an epoch will contain $\lceil n/k \rceil$ updates. An epoch essentially means that every point in the training data set has been seen exactly once.

Stochastic gradient-descent methods have much lower memory requirements than pure gradient-descent, because one is processing only a small sample of the data in each step. Although each update is more noisy, the sampled gradient can be computed much faster. Therefore, even though more updates are required, the overall process is much faster. Why does stochastic gradient descent work so well in machine learning? At its core, mini-batch methods are random sampling methods. One is trying to estimate the gradient of a loss function using a random subset of the data. At the very beginning of the gradient-descent, the parameter vector $\overline{w}$ is grossly incorrect. Therefore, using only a small subset of the data is often sufficient to estimate the direction of descent very well, and the updates of mini-batch stochastic gradient descent are almost as good as those obtained using the full data (but with a tiny fraction of the computational effort). This is what contributes to the significant improvement in running time. When the parameter vector $\overline{w}$ nears the optimal value during descent, the effect of sampling error is more significant. Interestingly, it turns out that this type of error is actually *beneficial* in machine learning applications because of an effect referred to as *regularization*! The reason has to do with the subtle differences between how optimization is used traditionally as opposed to how it is used in machine learning applications. This will be the subject of the discussion in the next section.

4.5.3 How Optimization in Machine Learning Is Different

There are some subtle differences in how optimization is used in machine learning from the way it is used in traditional optimization. An important difference is that traditional optimization focuses on learning the parameters so as to optimize the objective function as much as possible. However, in machine learning, there is a differentiation between the *training data* and the (roughly similar) unseen *test data*. For example, an entrepreneur may build an optimization model based on a history of how the *independent attributes* (like forecasting indicators) relate to the *dependent variable* (like actual sales) by minimizing the squared error of prediction of the dependent variable. The assumption is that the entrepreneur is using this model to make future predictions that are not yet known, and therefore the model can only be evaluated in retrospect on new data. Predicting the training data accurately does not always help one predict unseen test data more accurately. The general rule is that the optimized model will almost always predict the dependent variable of the training data more accurately than that of the test data (since it was directly used in modeling). This difference results in some critical design choices for optimization algorithms.

Consider the example of linear regression, where one will often have training examples $(\overline{X}_1, y_1) \ldots (\overline{X}_n, y_n)$ and a separate set of test examples $(\overline{Z}_1, y'_1) \ldots (\overline{Z}_t, y'_t)$. The labels of the test examples are unavailable in real-world applications at the time they are predicted.

In practice, they often become available only in *retrospect*, when the true accuracy of the machine learning algorithm can be computed. Therefore, the labels of the test examples cannot be made available during training. *In machine learning, one only cares about accuracy on the unseen test examples rather than training examples.* It is possible for excellently designed optimization methods to perform very well on the training data, but have disastrously poor results on the test data. This separation between training and test data is also respected during benchmarking of machine learning algorithms by creating simulated training and test data sets from a single labeled data set. In order to achieve this goal, one simply hides a part of the labeled data, and refers to the available part as the training data and the remainder as the test data. After building the model on the training data, one evaluates the performance of the model *on the test data, which was never seen during the training phase.* This is a key difference from traditional optimization, because the model is constructed using a particular data set; yet, a different (but similar) data set is used to evaluate performance of the optimization algorithm. This difference is crucial because models that perform very well on the training data might not perform very well on the test data. In other words, the model needs to *generalize* well to unseen test data. When a model performs very well on the training data, but does not perform very well on the unseen test data, the phenomenon is referred to as *overfitting*.

In order to understand this point, consider a case where one has a 4-dimensional data set of individuals, in which the four attributes x_1, x_2, x_3, and x_4 correspond to arm span, number of freckles, length of hair, and the length of nails. The arm span is defined as the maximum distance between fingertips when an individual holds their arms out wide. The target attribute is the height of the individual. The arm span is known to be almost equal to the height of an individual (with minor variations across races, genders, and individuals), although the goal of the machine learning application is to *infer* this fact in a data-driven manner. The predicted height of the individual is modeled by the linear function $\hat{y} = w_1x_1 + w_2x_2 + w_3x_3 + w_4x_4 + w_5$ for the purposes of prediction. The *best-fit* coefficients $w_1 \ldots w_5$ can be learned in a data-driven manner by minimizing the squared loss between predicted $\hat{y}$ and observed y. One would expect that the height of an individual is highly correlated with their arm span, but the number of freckles and lengths of hair/nails are not similarly correlated. As a result, one would typically expect w_1x_1 to make most of the contribution to the prediction, and the other three attributes would contribute very little (or noise). If the number of training examples is large, one would typically learn values of w_i that show this type of behavior. However, a different situation arises, if the number of training examples is small. For a problem with five parameters $w_1 \ldots w_5$, one needs at least 5 training examples to avoid a situation where an infinite number of solutions to the parameter vector exist (typically with zero error *on the training data*). This is because a system of equations of the form $y = w_1x_1 + w_2x_2 + w_3x_3 + w_4x_4 + w_5$ has an infinite number of equally good best-fit solutions if there are fewer equations than the number of variables. In fact, one can often find at least one solution in which w_1 is 0, and the squared error $(y - \sum_{i=1}^{4} w_ix_i - w_5)^2$ takes on its lowest possible value of zero on the training data. In spite of this fact, the error in the test data will typically be very high. Consider an example of a training set containing the following three data points:

Arm Span (inches)	Freckles (number)	Hair Length (inches)	Nail Length (inches)	Height (inches)
61	2	3	0.1	59
40	0	4	0.5	40
68	0	10	1.0	70

In this case, setting w_1 to 1 and all other coefficients to 0 is the "correct" solution, based on what is likely to happen *over an infinite number of training examples*. Note that this solution does not provide zero training error on this specific training data set, because there are always empirical variations across individuals. If we had an large number of examples (unlike the case of this table), it would also be possible for a model to learn this behavior well with a loss function that penalizes only the squared errors of predictions. However, with only three training examples, many other solutions exist that have zero training error. For example, setting $w_1 = 0$, $w_2 = 7$, $w_3 = 5$, $w_4 = 0$, and $w_5 = 20$ provides zero error on the training data. Here, the arm span and the nail length are not used at all. At the same time, setting $w_1 = 0$, $w_2 = 21.5$, $w_3 = 0$, $w_4 = 60$, and $w_5 = 10$ also yields zero error on the training data. This solution does not use the arm span or the hair length. Furthermore, any convex combination of these coefficients also provides zero error on the training data. Therefore, an infinite number of solutions that use irrelevant attributes provide better training error than the natural and intuitive solution that uses arm span. This is primarily because of overfitting to the specific training data at hand; this solution will generalize poorly to unseen test data.

All machine learning applications are used on unseen test data in real settings; therefore, it is unacceptable to have models that perform well on training data but perform poorly on test data. *Poor generalization is a result of models adapting to the quirks and random nuances of a specific training data set; it is likely to occur when the training data is small.* When the number of training instances is fewer than the number of features, an infinite number of equally "good" solutions exist. In such cases, poor generalization is almost inevitable unless steps are taken to avoid this problem. Therefore, there are a number of special properties of optimization in machine learning:

1. In traditional optimization, one optimizes the parameters as much as possible to improve the objective function. However, in machine learning, optimizing the parameter vector beyond a certain point often leads to overfitting. One approach is to hide a portion of the labeled data (which is referred to as the *held-out data*), perform the optimization, and always calculate the *out-of-sample accuracy* on this held-out data. Towards the end of the optimization process, the accuracy on the out-of-sample data begins to rise (even though the loss on the training data might continue to reduce). At this point, the learning is terminated. Therefore, the criterion for termination is different from that in traditional optimization.

2. While stochastic gradient-descent methods have lower accuracy than gradient-descent methods on training data (because of a sampling approximation), they often perform comparably (or even better) on the test data. This is because the random sampling of training instances during optimization reduces overfitting.

3. The objective function is sometimes modified by penalizing the squared norms of weight vectors. While the unmodified objective function is the most direct surrogate for the performance on the training data, the penalized objective function performs better on the out-of-sample test data. Concise parameter vectors with smaller squared norms are less prone to overfitting. This approach is referred to as *regularization*.

These differences between traditional optimization and machine learning are important because they affect the design of virtually every optimization procedure in machine learning.

4.5.4 Tuning Hyperparameters

As we have already seen, the learning process requires us to specify a number of hyperparameters such as the learning rate, the weight of regularization, and so on. The term "hyperparameter" is used to specifically refer to the parameters regulating the design of the model (like learning rate and regularization), and they are different from the more fundamental parameters such as the weights of the linear regression model. Machine learning always uses a two-tiered organization of parameters in the model, in which primary model parameters like weights are optimized with computational learning algorithms (e.g., stochastic gradient descent) only after fixing the hyperparameters either manually or with the use of a *tuning* phase. Here, it is important to note that the hyperparameters should not be tuned using the same data used for gradient descent. Rather, a portion of the data is held out as *validation data*, and the performance of the model is tested on the validation set with various choices of hyperparameters. This type of approach ensures that the tuning process does not overfit to the training data set.

The main challenge in hyperparameter optimization is that different combinations of hyperparameters need to be tested for their performance. The most well-known technique is *grid search*, in which all combinations of selected values of the hyperparameters are tested in order to determine the optimal choice. One issue with this procedure is that the number of hyperparameters might be large, and the number of points in the grid increases *exponentially* with the number of hyperparameters. For example, if we have 5 hyperparameters, and we test 10 values for each hyperparameter, the training procedure needs to be executed $10^5 = 100000$ times to test its accuracy. Therefore, a commonly used trick is to first work with coarse grids. Later, when one narrows down to a particular range of interest, finer grids are used. One must be careful when the optimal hyperparameter selected is at the edge of a grid range, because one would need to test beyond the range to see if better values exist.

The testing approach may at times be too expensive even with the coarse-to-fine-grained process. In some cases, it makes sense to randomly sample the hyperparameters uniformly within the grid range [14]. As in the case of grid ranges, one can perform multi-resolution sampling, where one first samples in the full grid range. One then creates a new set of grid ranges that are geometrically smaller than the previous grid ranges and centered around the optimal parameters from the previously explored samples. Sampling is repeated on this smaller box and the entire process is iteratively repeated multiple times to refine the parameters.

Another key point about sampling many types of hyperparameters is that the *logarithms* of the hyperparameters are sampled uniformly rather than the hyperparameters themselves. Two examples of such parameters include the regularization rate and the learning rate. For example, instead of sampling the learning rate α between 0.1 and 0.001, we first sample $\log_{10}(\alpha)$ uniformly between -1 and -3, and then exponentiate it as a power of 10. It is more common to search for hyperparameters in the logarithmic space, although there are some hyperparameters that should be searched for on a uniform scale.

4.5.5 The Importance of Feature Preprocessing

Vastly varying sensitivities of the loss function to different parameters tend to hurt the learning, and this aspect is controlled by the scale of the features. Consider a model in which a person's wealth is modeled as a linear function of the age x_1 (in the range $[0, 100]$), and the number of years of college education x_2 (in the range $[0, 10]$) as follows:

$$y = w_1 x_1^2 + w_2 x_2^2 \tag{4.17}$$

In such a case, the partial derivative $\frac{\partial y}{\partial w_1} = x_1^2$ and $\frac{\partial y}{\partial w_2} = x_2^2$ will show up as multiplicative terms in the components of the error gradient with respect to w_1 and w_2, respectively. Since x_1^2 is usually much larger than x_2^2 (and often by a factor of 100), the components of the error gradient with respect to w_1 will typically be much greater in magnitude than those with respect to w_2. Often, small steps along w_2 will lead to large steps along w_1 (and therefore an overshooting of the optimal value along w_1). Note that the sign of the gradient component along the w_1 direction will often keep flipping in successive steps to compensate for the overshooting along the w_1 direction after large steps. In practice, this leads to a back-and-forth "bouncing" behavior along the w_1 direction and tiny (but consistent) progress along the w_2 direction. As a result, convergence will be very slow. This type of behavior is discussed in greater detail in the next chapter. Therefore, it is often helpful to have features with similar variance. There are two forms of feature preprocessing used in machine learning algorithms:

1. *Mean-centering:* In many models, it can be useful to mean-center the data in order to remove certain types of bias effects. Many algorithms in traditional machine learning (such as principal component analysis) also work with the assumption of mean-centered data. In such cases, a vector of column-wise means is subtracted from each data point.

2. *Feature normalization:* A common type of normalization is to divide each feature value by its standard deviation. When this type of feature scaling is combined with mean-centering, the data is said to have been *standardized*. The basic idea is that each feature is presumed to have been drawn from a *standard* normal distribution with zero mean and unit variance.

   Min-max normalization is useful when the data needs to be scaled in the range $(0, 1)$. Let min_j and max_j be the minimum and maximum values of the jth attribute. Then, each feature value x_{ij} for the jth dimension of the ith point is scaled by min-max normalization as follows:

$$x_{ij} \Leftarrow \frac{x_{ij} - min_j}{max_j - min_j} \tag{4.18}$$

Feature normalization avoids ill-conditioning and ensures much smoother convergence of gradient-descent methods.

4.6 Computing Derivatives with Respect to Vectors

In typical optimization models encountered in machine learning, one is differentiating scalar objective functions (or even vectored quantities) with respect to vectors of parameters. This is because the loss function $J(\overline{w})$ is often a function of a vector of parameters $\overline{w}$. Rather than having to write out large numbers of partial derivatives with respect to each component of the vector, it is often convenient to represent such derivatives in *matrix calculus* notation. In the matrix calculus notation, one can compute a derivative of a scalar, vector, or matrix with respect to another scalar, vector, or matrix. The result might be a scalar, vector, matrix, or tensor; the final result can often be compactly expressed in terms of the vectors/matrices in the partial derivative (and therefore one does not have to tediously compute them in elementwise form). In this book, we will restrict ourselves to computing the derivatives of scalars/vectors with respect to other scalars/vectors. Occasionally, we will consider derivatives of scalars with respect to matrices. The result is always a scalar, vector,

or matrix. Being able to differentiate blocks of variables with respect to other blocks is useful from the perspective of brevity and quick computation. Although the field of matrix calculus is very broad, we will focus on a few important identities, which are useful for addressing the vast majority of machine learning problems one is likely to encounter in practice.

4.6.1 Matrix Calculus Notation

The simplest (and most common) example of matrix calculus notation arises during the computation of gradients. For example, consider the gradient-descent update for multivariate optimization problems, as discussed in the previous section:

$$\overline{w} \Leftarrow \overline{w} - \alpha \nabla J$$

An equivalent notation for the gradient ∇J is the matrix-calculus notation $\frac{\partial J(\overline{w})}{\partial \overline{w}}$. This notation is a scalar-to-vector derivative, which always returns a vector. Therefore, we have the following:

$$\nabla J = \frac{\partial J(\overline{w})}{\partial \overline{w}} = \left[\frac{\partial J(\overline{w})}{\partial w_1} \cdots \frac{\partial J(\overline{w})}{\partial w_d} \right]^T$$

Here, it is important to note that there is some convention-centric ambiguity in the treatments of matrix calculus by various communities as to whether the derivative of a scalar with respect to a column vector is a row vector or whether it is a column vector. Throughout this book, we use the convention that the derivative of a scalar with respect to a column vector is also a column vector. This convention is referred to as the *denominator layout* (although the numerator layout is more common in which the derivative is a row vector). We use the denominator layout because it frees us from the notational clutter of always having to transpose a row vector into a column vector in order to perform gradient descent updates on $\overline{w}$ (which are extremely common in machine learning). Indeed, the choice of using the numerator layout and denominator layout in different communities is often regulated by these types of notational conveniences. Therefore, we can directly write the update in matrix calculus notation as follows:

$$\overline{w} \Leftarrow \overline{w} - \alpha \left[\frac{\partial J(\overline{w})}{\partial \overline{w}} \right]$$

The matrix calculus notation also allows derivatives of vectors with respect to vectors. Such a derivative results in a matrix, referred to as the *Jacobian*. Jacobians arise frequently when computing the gradients of recursively nested multivariate functions; a specific example is the case of multilayer neural networks (cf. Chapter 11). For example, the derivative of an m-dimensional column vector $\overline{h} = [h_1, \ldots, h_m]^T$ with respect to a d-dimensional column vector $\overline{w} = [w_1, \ldots, w_d]^T$ is a $d \times m$ matrix in the denominator layout. The (i, j)th entry of this matrix is the derivative of h_j with respect to w_i:

$$\left[\frac{\partial \overline{h}}{\partial \overline{w}} \right]_{ij} = \frac{\partial h_j}{\partial w_i} \tag{4.19}$$

The (i, j)th element of the Jacobian is always $\frac{\partial h_i}{\partial w_j}$, and therefore it is the transpose of the matrix $\frac{\partial \overline{h}}{\partial \overline{w}}$ shown in Equation 4.19.

Another useful derivative that arises frequently in different types of matrix factorization is the derivative of a scalar objective function J with respect to an $m \times n$ matrix W. In the

denominator layout, the result inherits the shape of the matrix in the denominator. The (i, j)th entry of the derivative is simply the derivative of J with respect to the (i, j)th entry in W.

$$\left[\frac{\partial J}{\partial W}\right]_{ij} = \frac{\partial J}{\partial W_{ij}} \tag{4.20}$$

A review of matrix calculus notations and conventions is provided in Table 4.1.

4.6.2 Useful Matrix Calculus Identities

In this section, we will introduce a number of matrix calculus identities that are used frequently in machine learning. A common expression that arises commonly in machine learning is of the following form:

$$F(\overline{w}) = \overline{w}^T A \overline{w} \tag{4.21}$$

Here, A is a $d \times d$ symmetric matrix of constant values and $\overline{w}$ is a d-dimensional column vector of optimization variables. Note that this type of objective function occurs in virtually every convex quadratic loss function like least-squares regression and in the (dual) support-vector machine. In such a case, the gradient $\nabla F(\overline{w})$ can be written as follows:

$$\nabla F(\overline{w}) = \frac{\partial F(\overline{w})}{\partial \overline{w}} = 2A\overline{w} \tag{4.22}$$

The algebraic similarity of the derivative to the scalar case is quite noticeable. The reader is encouraged to work out each element-wise partial derivative and verify that the above expression is indeed correct. Note that $\nabla F(\overline{w})$ is a column vector.

Another common objective function $G(\overline{w})$ in machine learning is the following:

$$G(\overline{w}) = \overline{b}^T B \overline{w} = \overline{w}^T B^T \overline{b} \tag{4.23}$$

Here, B is an $n \times d$ matrix of constant values and $\overline{w}$ is a d-dimensional column vector of optimization variables. Furthermore, $\overline{b}$ is an n-dimensional constant vector that does not depend on $\overline{w}$. Therefore, this is a linear function in $\overline{w}$ and all components of the gradient are constants. The values $\overline{b}^T B \overline{w}$ and $\overline{w}^T B^T \overline{b}$ are the same because the transposition of a scalar is the same scalar. In such cases, the gradient of $G(\overline{w})$ is computed as follows:

$$\nabla G(\overline{w}) = \frac{\partial G(\overline{w})}{\partial \overline{w}} = B^T \overline{b} \tag{4.24}$$

In this case, every component of the gradient is a constant. We leave the proofs of these results as a practice exercise:

Problem 4.6.1 *Let $A = [a_{ij}]$ be a symmetric $d \times d$ matrix of constant values, $B = [b_{ij}]$ be an $n \times d$ matrix of constant values, $\overline{w}$ be a d-dimensional column vector of optimization variables, and $\overline{b}$ be an n-dimensional column vector of constants. Let $F(\overline{w}) = \overline{w}^T A \overline{w}$ and let $G(\overline{w}) = \overline{b}^T B \overline{w}$. Show using component-wise partial derivatives that $\nabla F(\overline{w}) = 2A\overline{w}$ and $\nabla G(\overline{w}) = B^T \overline{b}$.*

The above practice exercise would require one to expand each expression in terms of the scalar values in the matrices and vectors. One can then appreciate the compactness of the matrix calculus approach for quick computation. We provide a list of the commonly used identities in Table 4.2. Many of these identities are useful in machine learning models.

Table 4.1: Matrix calculus operations in numerator and denominator layouts

Derivative of:	**with respect to:**	Output size	ith or (i,j)th element
Scalar J	Scalar x	Scalar	$\frac{\partial J}{\partial x}$
Column vector $\overline{h}$ in m dimensions	Scalar x	Column vector in m dimensions	$\left[\frac{\partial \overline{h}}{\partial x}\right]_i = \frac{\partial h_i}{\partial x}$
Scalar J	Column vector $\overline{w}$ in d dimensions	Row vector in d dimensions	$\left[\frac{\partial J}{\partial \overline{w}}\right]_i = \frac{\partial J}{\partial w_i}$
Column vector $\overline{h}$ in m dimensions	Column vector $\overline{w}$ in d dimensions	$m \times d$ matrix	$\left[\frac{\partial \overline{h}}{\partial \overline{w}}\right]_{ij} = \frac{\partial h_i}{\partial w_j}$
Scalar J	$m \times n$ matrix W	$n \times m$ matrix	$\left[\frac{\partial J}{\partial W}\right]_{ij} = \frac{\partial J}{\partial W_{ji}}$

(a) Numerator layout

Derivative of:	**with respect to:**	Output size	ith or (i,j)th element
Scalar J	Scalar x	Scalar	$\frac{\partial J}{\partial x}$
Column vector $\overline{h}$ in m dimensions	Scalar x	Row vector in m dimensions	$\left[\frac{\partial \overline{h}}{\partial x}\right]_i = \frac{\partial h_i}{\partial x}$
Scalar J	Column vector $\overline{w}$ in d dimensions	Column vector in d dimensions	$\left[\frac{\partial J}{\partial \overline{w}}\right]_i = \frac{\partial J}{\partial w_i}$
Column vector $\overline{h}$ in m dimensions	Column vector $\overline{w}$ in d dimensions	$d \times m$ matrix	$\left[\frac{\partial \overline{h}}{\partial \overline{w}}\right]_{ij} = \frac{\partial h_j}{\partial w_i}$
Scalar J	$m \times n$ matrix W	$m \times n$ matrix	$\left[\frac{\partial J}{\partial W}\right]_{ij} = \frac{\partial J}{\partial W_{ij}}$

(b) Denominator layout

Table 4.2: List of common matrix calculus identities in denominator layout. A is a constant $d \times d$ matrix, B is a constant $n \times d$ matrix, and $\overline{b}$ is a constant n-dimensional vector independent of the parameter vector $\overline{w}$. C is a $k \times d$ matrix

	Objective J	**Derivative of J with respect to $\overline{w}$**
(i)	$\overline{w}^T A\overline{w}$	$2A\overline{w}$ (symmetric A) $(A + A^T)\overline{w}$ (asymmetric A)
(ii)	$\overline{b}^T B\overline{w}$ or $\overline{w}^T B^T \overline{b}$	$B^T \overline{b}$
(iii)	$\|\|B\overline{w} + \overline{b}\|\|^2$	$2B^T(B\overline{w} + \overline{b})$
(iv)	$f(g(\overline{w}))$ [$g(\overline{w})$ is scalar: example below]	$f'(g(\overline{w}))\nabla_w g(\overline{w})$
(v)	$f(\overline{w} \cdot \overline{a})$ [Example $g(\overline{w}) = \overline{w} \cdot \overline{a}$ of above]	$f'(\overline{w} \cdot \overline{a})\overline{a}$

(a) Scalar-to-vector derivatives

	Vector $\overline{h}$	**Derivative of $\overline{h}$ with respect to $\overline{w}$**
(i)	$\overline{h} = C\overline{w}$	C^T
(ii)	$\overline{h} = F(\overline{w})$ [$F(\cdot)$ is elementwise function]	Diagonal matrix with (i,i)th entry containing partial derivative of ith component of $F(\overline{w})$ w.r.t. w_i
(iii)	Product-of-variables identity $\overline{h} = f_s(\overline{w})\overline{x}$ [$f_s(\overline{w})$ is vector-to-scalar function]	$\frac{\partial f_s(\overline{w})}{\partial \overline{w}}\overline{x}^T + f_s(\overline{w})\frac{\partial \overline{x}}{\partial \overline{w}}$

(b) Vector-to-vector derivatives

Since it is common to compute the gradient with respect to a column vector of parameters, all these identities represent the derivatives with respect to a column vector. Note that Table 4.2(b) represent some simple vector-to-vector derivatives, which always lead to the transpose of the Jacobian. Beyond these commonly used identities, a full treatment of matrix calculus is beyond the scope of the book, although interested readers are referred to [20].

4.6.2.1 Application: Unconstrained Quadratic Programming

In *quadratic programming*, the objective function contains a quadratic term of the form $\overline{w}^T A\overline{w}$, a linear term $\overline{b}^T \overline{w}$, and a constant. An unconstrained quadratic program has the following form:

$$\text{Minimize}_{\overline{w}} \frac{1}{2}\overline{w}^T A\overline{w} + \overline{b}^T \overline{w} + c$$

Here, we assume that A is a *positive definite* $d \times d$ matrix, $\overline{b}$ is a d-dimensional column vector, c is a scalar constant, and the optimization variables are contained in the d-dimensional column vector $\overline{w}$. An unconstrained quadratic program is a direct generalization of 1-dimensional quadratic functions like $\frac{1}{2}ax^2 + bx + c$. Note that a minimum exists at $x = -b/a$ for 1-dimensional quadratic functions when $a > 0$, and a minimum exists for multidimensional quadratic functions when A is positive definite.

The two terms in the objective function can be differentiated with respect to $\overline{w}$ by using the identities (i) and (ii) in Table 4.2(a). Since the matrix A is positive definite, it follows that the Hessian A is positive definite irrespective of the value of $\overline{w}$. Therefore, the objective function is strictly convex, and setting the gradient to zero is a necessary and

sufficient condition for minimization of the objective function. Using the identities (i) and (ii) of Table 4.2(a), we obtain the following optimality condition:

$$A\overline{w} + \overline{b} = \overline{0}$$

Therefore, we obtain the solution $\overline{w} = -A^{-1}\overline{b}$. Note that this is a direct generalization of the solution for the 1-dimensional quadratic function. In the event that A is singular, a solution is not guaranteed even when A is positive semidefinite. For example, when A is the zero matrix, the objective function becomes linear with no minimum. When A is positive semidefinite, it can be shown that a minimum exists if and only if $\overline{b}$ lies in the column space of A (see Exercise 8).

4.6.2.2 Application: Derivative of Squared Norm

A special case of unconstrained quadratic programming is the norm of a vector that is itself a linear function of another vector (with an additional constant offset). Such a problem arises in least-squares regression, which is known to have a closed form solution (cf. Section 4.7) like the quadratic program of the previous section. This particular objective function has the following form:

$$\begin{aligned} J(\overline{w}) &= \|B\overline{w} + \overline{b}\|^2 \\ &= \overline{w}^T B^T B\overline{w} + 2\overline{b}^T B\overline{w} + \overline{b}^T \overline{b} \end{aligned}$$

Here, B is an $n \times d$ data matrix, $\overline{w}$ is a d-dimensional vector, and $\overline{b}$ is an n-dimensional vector. This form of the objective function arises frequently in least-squares-regression, where B is set to the observed data matrix D, and the constant vector $\overline{b}$ is set to the negative of the response vector $\overline{y}$. One needs to compute the gradient with respect to $\overline{w}$ in order to perform the updates.

We have expanded the squared norm in terms of matrix vector products above. The individual terms are of the same form as the results (i) and (ii) of Table 4.2(a). In such a case, we can compute the derivative of the squared norm with respect to $\overline{w}$ by substituting for the scalar-to-vector derivatives in results (i) and (ii) Table 4.2(a). Therefore, we obtain the following results:

$$\frac{\partial J(\overline{w})}{\partial \overline{w}} = 2B^T B\overline{w} + 2B^T \overline{b} \tag{4.25}$$

$$= 2B^T(B\overline{w} + \overline{b}) \tag{4.26}$$

This form of the gradient is used often in least-squares regression. Setting this gradient to zero yields the closed-form solution to least-squares regression (cf. Section 4.7).

4.6.3 The Chain Rule of Calculus for Vectored Derivatives

The chain rule of calculus is extremely useful for differentiating compositions of functions. In the univariate case with scalars, the rule is quite simple. For example, consider the case where the scalar objective J is a function of the scalar w as follows:

$$J = f(g(h(w))) \tag{4.27}$$

All of $f(\cdot)$, $g(\cdot)$, and $h(\cdot)$ are assumed to be scalar functions. In such a case, the derivative of J with respect to the scalar w is simply $f'(g(h(w)))g'(h(w))h'(w)$. This rule is referred

to as the univariate chain rule of differential calculus. Note that the order of multiplication does not matter because scalar multiplication is commutative.

Similarly, consider the case where you have the following functions, where one of the functions is a vector-to-scalar function:

$$J = f(g_1(w), g_2(w), \ldots, g_k(w))$$

In such a case, the *multivariate chain rule* states that one can compute the derivative of J with respect to w as the sum of the products of the partial derivatives using all arguments of the function:

$$\frac{\partial J}{\partial w} = \sum_{i=1}^{k} \left[\frac{\partial J}{\partial g_i(w)}\right] \left[\frac{\partial g_i(w)}{\partial w}\right]$$

One can generalize *both* of the above results into a single form by considering the case where the functions are vector-to-vector functions. Note that vector-to-vector derivatives are matrices, and therefore we will be multiplying matrices together instead of scalars. Surprisingly, very large classes of machine learning algorithms perform the repeated composition of only two types of functions, which are shown in Table 4.2(b). *Unlike the case of the scalar chain rule, the order of multiplication is important when dealing with matrices and vectors.* In a composition function, the derivative of the argument (inner level variable) is always pre-multiplied with the derivative of the function (outer level variable). In many cases, the order of multiplication is self-evident because of the size constraints associated with matrix multiplication. We formally define the vectored chain rule as follows:

Theorem 4.6.1 (Vectored Chain Rule) *Consider a composition function of the following form:*

$$\overline{o} = F_k(F_{k-1}(\ldots F_1(\overline{x})))$$

Assume that each $F_i(\cdot)$ takes as input an n_i-dimensional column vector and outputs an n_{i+1}-dimensional column vector. Therefore, the input $\overline{x}$ is an n_1-dimensional vector and the final output $\overline{o}$ is an n_{k+1}-dimensional vector. For brevity, denote the vector output of $F_i(\cdot)$ by $\overline{h}_i$. Then, the vectored chain rule asserts the following:

$$\underbrace{\left[\frac{\partial \overline{o}}{\partial \overline{x}}\right]}_{n_1 \times n_{k+1}} = \underbrace{\left[\frac{\partial \overline{h}_1}{\partial \overline{x}}\right]}_{n_1 \times n_2} \underbrace{\left[\frac{\partial \overline{h}_2}{\partial \overline{h}_1}\right]}_{n_2 \times n_3} \cdots \underbrace{\left[\frac{\partial \overline{h}_{k-1}}{\partial \overline{h}_{k-2}}\right]}_{n_{k-1} \times n_k} \underbrace{\left[\frac{\partial \overline{o}}{\partial \overline{h}_{k-1}}\right]}_{n_k \times n_{k+1}}$$

It is easy to see that the size constraints of matrix multiplication are respected in this case.

4.6.3.1 Useful Examples of Vectored Derivatives

In the following, we provide some examples of vectored derivatives that are used frequently in machine learning. Consider the case where the function $g(\cdot)$ has a d-dimensional vector argument and its output is scalar. Furthermore, the function $f(\cdot)$ is a scalar-to-scalar function.

$$J = f(g(\overline{w}))$$

In such a case, we can apply the vectored chain rule to obtain the following:

$$\nabla J = \frac{\partial J}{\partial \overline{w}} = \nabla g(\overline{w}) \underbrace{f'(g(\overline{w}))}_{\text{scalar}} \tag{4.28}$$

In this case, the order of multiplication does not matter, because one of the factors in the product is a scalar. Note that this result is used frequently in machine learning, because many loss-functions in machine learning are computed by applying a scalar function $f(\cdot)$ to the dot product of $\overline{w}$ with a training point $\overline{a}$. In other words, we have $g(\overline{w}) = \overline{w} \cdot \overline{a}$. Note that $\overline{w} \cdot \overline{a}$ can be written as $\overline{w}^T(I)\overline{a}$, where I represents the identity matrix. This is in the form of one of the matrix identities of Table 4.2(a) [see identity (ii)]. In such a case, one can use the chain rule to obtain the following:

$$\frac{\partial J}{\partial \overline{w}} = \underbrace{[f'(g(\overline{w}))]}_{\text{scalar}} \overline{a} \tag{4.29}$$

This result is extremely useful, and it can be used for computing the derivatives of many loss functions like least-squares regression, SVMs, and logistic regression. The vector $\overline{a}$ is simply replaced with the vector of the training point at hand. The function $f(\cdot)$ defines the specific form of the loss function for the model at hand. We have listed these identities as results (iv) and (v) of Table 4.2(a).

Table 4.2(b) contains a number of useful derivatives of vector-to-vector functions. The first is the linear transformation $\overline{h} = C\overline{w}$, where C is a matrix that does not depend on the parameter vector $\overline{w}$. The corresponding vector-to-vector derivative of $\overline{h}$ with respect to $\overline{w}$ is C^T [see identity (i) of Table 4.2(b)]. This type of transformation is used commonly in *linear layers of feed-forward neural networks*. Another common vector-to-vector function is the element-wise function $F(\overline{w})$, which is also used in neural networks (in the form of *activation functions*). In this case, the corresponding derivative is a diagonal matrix containing the element-wise derivatives as shown in the second identity of Table 4.2(b).

Finally, we consider a generalization of the *product identity* in differential calculus. Instead of differentiating the product of two scalar variables, we consider the product of a scalar and a vector variable. Consider the relationship $\overline{h} = f_s(\overline{w})\overline{x}$, which is the product of a vector and a scalar. Here, $f_s(\cdot)$ is a vector-to-scalar function and $\overline{x}$ is a column vector that depends on $\overline{w}$. In such a case, the derivative of $\overline{h}$ with respect to $\overline{w}$ is the matrix $\frac{\partial f_s(\overline{w})}{\partial \overline{w}}\overline{x}^T + f_s(\overline{w})\frac{\partial \overline{x}}{\partial \overline{w}}$ [see identity (iii) of Table 4.2(b)]. Note that the first term is the outer product of the two vectors $\frac{\partial f_s(\overline{w})}{\partial \overline{w}}$ and $\overline{x}$, whereas the second term is a scalar multiple of a vector-to-vector derivative.

4.7 Linear Regression: Optimization with Numerical Targets

Linear regression is also referred to as least-squares regression, because it is usually paired with a least-squares objective function. Least-squares regression was introduced briefly in Section 2.8 of Chapter 2 in order to provide an optimization-centric view of solving systems of equations. A more natural application of least-squares regression is to model the dependence of a target variable on the feature variables. We have n pairs of observations $(\overline{X}_i, y_i)$ for $i \in \{1 \ldots n\}$. The target y_i is predicted using $\hat{y}_i \approx \overline{W} \cdot \overline{X}_i^T$. The circumflex on top of $\hat{y}_i$ indicates that it is a predicted value. Here, $\overline{W} = [w_1 \ldots w_d]^T$ is a d-dimensional column vector of optimization parameters.

Each vector $\overline{X}_i$ is referred to as the set of independent variables or *regressors*, whereas the variable y_i is referred to as the target variable, response variable, or *regressand*. Each $\overline{X}_i$ is a row vector, because it is common for data points to be represented as rows of data

matrices in machine learning. Therefore, the row vector $\overline{X}_i$ needs to be transposed before performing a dot product with the column vector $\overline{W}$. The vector $\overline{W}$ needs to be learned in a data driven manner, so that $\hat{y}_i = \overline{W} \cdot \overline{X}_i^T$ is as close to each y_i as possible. Therefore, we compute the loss $(y_i - \overline{W} \cdot \overline{X}_i^T)^2$ for each training data point, and then add up this losses over all points in order to create the objective function:

$$J = \frac{1}{2} \sum_{i=1}^{n} (y_i - \overline{W} \cdot \overline{X}_i^T)^2 \tag{4.30}$$

Once the vector $\overline{W}$ has been learned from the training data by optimizing the aforementioned objective function, the numerical value of the target variable of an unseen test instance $\overline{Z}$ (which is a d-dimensional row vector) can be predicted as $\overline{W} \cdot \overline{Z}^T$.

It is particularly convenient to write this objective function in terms of an $n \times d$ data matrix. The $n \times d$ data matrix D is created by stacking up the n rows $\overline{X}_1 \ldots \overline{X}_n$. Similarly, $\overline{y}$ is an n-dimensional column vector of response variables for which the ith entry is y_i. Note that $D\overline{W}$ is an n-dimensional column vector of *predictions* which should ideally equal the observed vector $\overline{y}$. Therefore, the vector of errors is given by $(D\overline{W} - \overline{y})$, and the squared norm of the error vector is the loss function. Therefore, the minimization loss function of least-squares regression may be written as follows:

$$J = \frac{1}{2} \|D\overline{W} - \overline{y}\|^2 = \frac{1}{2} [D\overline{W} - \overline{y}]^T [D\overline{W} - \overline{y}] \tag{4.31}$$

One can expand the above expression as follows:

$$J = \frac{1}{2} \overline{W}^T D^T D \overline{W} - \frac{1}{2} \overline{W}^T D^T \overline{y} - \frac{1}{2} \overline{y}^T D \overline{W} + \frac{1}{2} \overline{y}^T \overline{y} \tag{4.32}$$

It is easy to see that the above expression is convex, because $D^T D$ is the positive semidefinite Hessian in the quadratic term. This means that if we find a value of the vector $\overline{W}$ at which the gradient is zero (i.e., a critical point), it will be a global minimum of the objective function.

In order to compute the gradient of J with respect to $\overline{W}$, one can directly use the squared-norm result of Section 4.6.2.2 to yield the following:

$$\nabla J = D^T D \overline{W} - D^T \overline{y} \tag{4.33}$$

Setting the gradient to zero yields the following condition:

$$D^T D \overline{W} = D^T \overline{y} \tag{4.34}$$

Pre-multiplying both sides with $(D^T D)^{-1}$, one obtains the following:

$$\overline{W} = (D^T D)^{-1} D^T \overline{y} \tag{4.35}$$

Note that this formula is identical to the use of the left-inverse of D for solving a system of equations (cf. Section 2.8 of Chapter 2), and the derivation of Section 2.8 uses the normal equation rather than calculus. The problem of solving a system of equations is a special case of least-squares regression. When the system of equations has a feasible solution, the optimal solution has zero loss on the training data. In the case that the system is inconsistent, we obtain the best-fit solution.

How can one compute $\overline{W}$ efficiently, when $D^T D$ is invertible? This can be achieved via QR decomposition of matrix D as $D = QR$ (see end of Section 2.8.2), where Q is an $n \times d$ matrix with orthonormal columns and R is a $d \times d$ upper-triangular matrix. One can simply substitute $D = QR$ in Equation 4.34, and use $Q^T Q = I_d$ to obtain the following:

$$R^T R\overline{W} = R^T Q^T \overline{y} \tag{4.36}$$

Multiplying both sides with $(R^T)^{-1}$, one obtains $R\overline{W} = Q^T \overline{y}$. This triangular system of equations can be solved efficiently using back-substitution.

The above solution assumes that the matrix $D^T D$ is invertible. However, in cases where the number of data points is small, the matrix $D^T D$ might not be invertible. In such cases, infinitely many solutions exist to this system of equations, which will overfit the training data; such methods will not generalize easily to unseen test data. In such cases, regularization is important.

4.7.1 Tikhonov Regularization

The closed-form solution to the problem does not work in under-determined cases, where the number of optimization variables is greater than the number of points. One possible solution is to reduce the number of variables in the data by posing the problem as a constrained optimization problem. In other words, we could try to optimize the same loss function while posing the hard constraint that at most k values of w_i are non-zero. However, such a constrained optimization problem is hard to solve. A softer solution is to impose a small penalty on the absolute value of each w_i in order to discourage non-zero values of w_i. Therefore, the resulting loss function is as follows:

$$J = \frac{1}{2}\|D\overline{W} - \overline{y}\|^2 + \frac{\lambda}{2}\|\overline{W}\|^2 \tag{4.37}$$

Here, $\lambda > 0$ is the regularization parameter. By adding the squared norm penalty, we are encouraging each w_i to be small in magnitude, unless it is absolutely essential for learning. Note that the addition of the strictly convex term $\lambda\|\overline{W}\|^2$ to the convex least-squares regression loss function makes the regularized objective function *strictly* convex (see Lemma 4.3.6 on addition of convex and strictly convex functions). A strictly convex objective function has a unique optimal solution.

In order to solve the optimization problem, one can set the gradient of J to 0. The gradient of the added term $\lambda\|\overline{W}\|^2/2$ is $\lambda\overline{W}$, based on the discussion in Section 4.6.2.2. On setting the gradient of J to 0, we obtain the following modified condition:

$$(D^T D + \lambda I)\overline{W} = D^T \overline{y} \tag{4.38}$$

Pre-multiplying both sides with $(D^T D + \lambda I)^{-1}$, one obtains the following:

$$\overline{W} = (D^T D + \lambda I)^{-1} D^T \overline{y} \tag{4.39}$$

Here, it is important to note that $(D^T D + \lambda I)$ is always invertible for $\lambda > 0$, since the matrix is positive definite (see Problem 2.4.2 of Chapter 2). The resulting solution is regularized, and it generalizes much better to out-of-sample data. Because of the *push-through identity* (see Problem 1.2.13), the solution can also be written in the following alternative form:

$$\overline{W} = D^T (DD^T + \lambda I)^{-1} \overline{y} \tag{4.40}$$

4.7.1.1 Pseudoinverse and Connections to Regularization

A special case of Tikhonov regularization is the *Moore-Penrose pseudoinverse*, which is introduced in Section 2.8.1 of Chapter 2. The Moore-Penrose pseudoinverse D^+ of the matrix D is the limiting case of Tikhonov regularization in which $\lambda > 0$ is infinitesimally small:

$$D^+ = \lim_{\lambda \to 0+} (D^T D + \lambda I)^{-1} D^T = \lim_{\lambda \to 0+} D^T (DD^T + \lambda I)^{-1} \tag{4.41}$$

Therefore, one can simply write the solution $\overline{W}$ in terms of the Moore-Penrose pseudoinverse as $\overline{W} = D^+ \overline{y}$.

4.7.2 Stochastic Gradient Descent

In machine learning, it is rare to obtain a closed-form solution like Equation 4.39. In most cases, one uses (stochastic) gradient-descent updates of the following form:

$$\overline{W} \Leftarrow \overline{W} - \alpha \nabla J \tag{4.42}$$

One advantage of (stochastic) gradient descent is that it is an efficient solution both in terms of memory requirements and computational efficiency. In the case of least-squares regression, the update of Equation 4.42 can be instantiated as follows:

$$\overline{W} \Leftarrow \overline{W}(1 - \alpha\lambda) - \alpha D^T \underbrace{(D\overline{W} - \overline{y})}_{\text{Error vector } \overline{e}} \tag{4.43}$$

Here, $\alpha > 0$ is the learning rate. In order to implement the approach efficiently, one first computes the n-dimensional error vector $\overline{e} = (D\overline{W} - \overline{y})$, which is marked in the above equation. Subsequently, the d-dimensional vector $D^T \overline{e}$ is computed for the update. Such an approach only requires matrix-vector multiplication, rather than requiring the materialization of the potentially large matrix $D^T D$.

One can also perform mini-batch stochastic gradient descent by selecting a subset of examples (rows) from the data matrix D. Let S be a set of training examples in the current mini-batch, where each example in S contains the feature-target pair in the form $(\overline{X}_i, y_i)$. Then, the gradient-descent update can be modified to the mini-batch update as follows:

$$\overline{W} \Leftarrow \overline{W}(1 - \alpha\lambda) - \alpha \sum_{(\overline{X}_i, y_i) \in S} \overline{X}_i^T \underbrace{(\overline{W} \cdot \overline{X}_i^T - y_i)}_{\text{Error value}} \tag{4.44}$$

Note that Equation 4.44 can be derived directly from Equation 4.43 by simply assuming that only the (smaller) matrix corresponding to the mini-batch is available at the time of the update.

4.7.3 The Use of Bias

It is common in machine learning to introduce an additional bias variable to account for unexplained constant effects in the targets. For example, consider the case in which the target variable is the temperature in a tropical city in Fahrenheit and the two feature variables respectively correspond to the number of days since the beginning of the year, and the number of minutes since midnight. The modeling $y_i = \overline{W} \cdot \overline{X}_i^T$ is bound to lead to large errors because of unexplained constant effects. For example, when both feature variables are 0, it corresponds to the New Year's eve. The temperature in a tropical city is bound to

be much higher than 0 on New Year's eve. However, the modeling $y_i = \overline{W} \cdot \overline{X}_i^T$ will always yield 0 as a predicted value. This problem can be avoided with the use of a bias variable b, so that the new model is $y_i = \overline{W} \cdot \overline{X}_i^T + b$. The bias variable absorbs the additional constant effects (i.e., bias specific to the city at hand) and it needs to be learned like the other parameters in $\overline{W}$. In such a case, it can be shown that the gradient-descent updates of Equation 4.44 are modified as follows:

$$\overline{W} \Leftarrow \overline{W}(1 - \alpha\lambda) - \alpha \sum_{(\overline{X}_i, y_i) \in S} \overline{X}_i^T \underbrace{(\overline{W} \cdot \overline{X}_i^T + b - y_i)}_{\text{Error value}}$$

$$b \Leftarrow b(1 - \alpha\lambda) - \alpha \sum_{(\overline{X}_i, y_i) \in S} \underbrace{(\overline{W} \cdot \overline{X}_i^T + b - y_i)}_{\text{Error value}}$$

It turns out that it is possible to achieve *exactly* the same effect as the above updates without changing the original (i.e., bias-free) model. The trick is to add an additional dimension to the training and test data with a constant value of 1. Therefore, one would have an additional $(d+1)$th parameter w_{d+1} in vector $\overline{W}$, and the target variable for $\overline{X} = [x_1 \ldots x_d]$ is predicted as follows:

$$\hat{y} = [\sum_{i=1}^{d} w_i x_i] + w_{i+1}(1)$$

It is not difficult to see that this is exactly the same prediction function as the one with bias. The coefficient w_{d+1} of this additional dimension is the bias variable b. Since the bias variable can be incorporated with a feature engineering trick, it will largely be omitted in most of the machine learning applications in this book. However, as a practical matter, it is very important to use the bias (in some form) in order to avoid undesirable constant effects.

4.7.3.1 Heuristic Initialization

Choosing a good initialization can sometimes be helpful in speeding up the updates. Consider a linear regression problem with an $n \times d$ data matrix D. In most cases, the number of training examples n is much greater than the number of features d. A simple approach for heuristic initialization is to select d randomly chosen training points and solve the $d \times d$ system of equations using any of the methods discussed in Chapter 2. Solving a system of linear equations is a special case of linear regression, and it is also much simpler. This provides a good initial starting point for the weight vector.

Problem 4.7.1 (Matrix Least-Squares) *Consider an $n \times d$ tall data matrix D and $n \times k$* **matrix** *Y of numerical targets. You want to find the $d \times k$ weight matrix W so that $\|DW - Y\|_F^2$ is as small as possible. Show that the optimal weight matrix is $W = (D^T D)^{-1} D^T Y$, assuming that D has linearly independent columns. Show that the left-inverse of a tall matrix D is the best least-squares solution to the matrix R satisfying the right-inverse relationship $DR \approx I_n$, and the resulting approximation of I_n is a projection matrix.*

4.8 Optimization Models for Binary Targets

Least-squares regression learns how to relate numerical feature variables (independent variables or regressor) to a numerical target (i.e., dependent variable or regressand). In many applications, the targets are discrete rather than real-valued. An example of such a target is

the color such as $\{Blue, Green, Red\}$. Note that there is no natural ordering between these targets, which is different from the case of numerical targets unless the target variable is binary.

A special case of discrete targets is the case in which the target variable y is binary and drawn from $\{-1, +1\}$. The instances with label $+1$ are referred to as *positive class instances*, and those with label -1 are referred to as *negative class instances*. For example, the feature variables in a cancer detection application might correspond to patient clinical measurements, and the class variable can be an indicator of whether or not the patient has cancer. In the binary-class case, we *can* impose an ordering between the two possible target values. In other words, we can pretend that the targets are numeric, and simply perform linear regression. This method is referred to as *least-squares classification*, which is discussed in the next section. Treating discrete targets as numerical values does have its disadvantages. Therefore, many alternative loss functions have been proposed for discrete (binary) data that avoid these disadvantages. Examples include the support vector machine and logistic regression. In the following, we will provide an overview of these models and their relationships with one another. While discussing these relationships, it will become evident that the ancient problem of least-squares regression serves as the parent model and the motivating force to all these (relatively recent) models for discrete-valued targets.

4.8.1 Least-Squares Classification: Regression on Binary Targets

In least-squares classification, linear regression is directly applied to binary targets. The $n \times d$ data matrix D still contains numerical values, and its rows $\overline{X}_1 \ldots \overline{X}_n$ are d-dimensional row vectors. However, the n-dimensional target vector $\overline{y} = [y_1 \ldots y_n]^T$ will only contain binary values drawn from -1 or $+1$. In least-squares classification, we pretend that the binary targets are real-valued. Therefore, we model each target as $y_i \approx \overline{W} \cdot \overline{X}_i^T$, where $\overline{W} = [w_1, \ldots, w_d]^T$ is a column vector containing the weights. We set up the same squared loss function as least-squares regression by treating binary targets as special cases of numerical targets. This results in the same closed-form solution for $\overline{W}$:

$$\overline{W} = (D^T D + \lambda I)^{-1} \overline{y} \tag{4.45}$$

Even though $\overline{W} \cdot \overline{X}_i^T$ yields a real-valued prediction for instance $\overline{X}_i$ (like regression), it makes more sense to view the hyperplane $\overline{W} \cdot \overline{X}^T = 0$ as a *separator* or *modeled decision boundary*, where any instance $\overline{X}_i$ with label $+1$ will satisfy $\overline{W} \cdot \overline{X}_i^T > 0$, and any instance with label -1 will satisfy $\overline{W} \cdot \overline{X}_i^T < 0$. Because of the way in which the model has been trained, most *training* points will align themselves on the two sides of the separator, so that the sign of the training label y_i matches the sign of $\overline{W} \cdot \overline{X}_i^T$. An example of a two-class data set in two dimensions is illustrated in Figure 4.8 in which the two classes are denoted by '+' and '*', respectively. In this case, it is evident that the value of $\overline{W} \cdot \overline{X}_i^T = 0$ is true only for points on the separator. The training points on the two sides of the separator satisfy either $\overline{W} \cdot \overline{X}_i^T < 0$ or $\overline{W} \cdot \overline{X}_i^T > 0$. The separator $\overline{W} \cdot \overline{X}^T = 0$ between the two classes is the modeled decision boundary. Note that some data distributions might not have the kind of neat separability as shown in Figure 4.8. In such cases, one either needs to live with errors or use feature transformation techniques to create linear separability. These techniques (such as *kernel methods*) are discussed in Chapter 9.

Once the weight vector $\overline{W}$ has been learned in the training phase, the classification is performed on an unseen test instance $\overline{Z}$. Since the test instance $\overline{Z}$ is a row vector, whereas

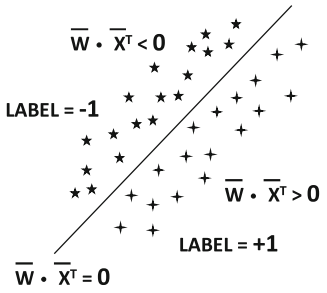


Figure 4.8: An example of linear separation between two classes

$\overline{W}$ is a column vector, the test instance needs to be transposed before computing the dot product between $\overline{W}$ and $\overline{Z}^T$. This dot product yields a real-valued prediction, which is converted to a binary prediction with the use of sign function:

$$\hat{y} = \text{sign}\{\overline{W} \cdot \overline{Z}^T\} \tag{4.46}$$

In effect, the model learns a linear hyperplane $\overline{W} \cdot \overline{X}^T = 0$ separating the positive and negative classes. All test instances for which $\overline{W} \cdot \overline{Z}^T > 0$ are predicted to belong to the positive class, and all instances for which $\overline{W} \cdot \overline{Z}^T < 0$ are predicted to belong to the negative class.

As in the case of real-valued targets, one can also use mini-batch stochastic gradient-descent for regression on binary targets. Let S be a mini-batch of pairs $(\overline{X}_i, y_i)$ of feature variables and targets. Each $\overline{X}_i$ is a row of the data matrix D and y_i is a target value drawn from $\{-1, +1\}$. Then, the mini-batch update for least-squares classification is identical to that of least-squares regression:

$$\overline{W} \Leftarrow \overline{W}(1 - \alpha\lambda) - \alpha \sum_{(\overline{X}_i, y_i) \in S} \overline{X}_i^T (\overline{W} \cdot \overline{X}_i^T - y_i) \tag{4.47}$$

Here, $\alpha > 0$ is the learning rate, and $\lambda > 0$ is the regularization parameter. Note that this update is *identical* to that in Equation 4.44. However, since each target y_i is drawn from $\{-1, +1\}$, an alternative approach also exists for writing the targets by using the fact that $y_i^2 = 1$. This alternative form of the update is as follows:

$$\begin{aligned}\overline{W} &\Leftarrow \overline{W}(1 - \alpha\lambda) - \alpha \sum_{(\overline{X}_i, y_i) \in S} \underbrace{y_i^2}_{1} \overline{X}_i^T (\overline{W} \cdot \overline{X}_i^T - y_i) \\ &= \overline{W}(1 - \alpha\lambda) - \alpha \sum_{(\overline{X}_i, y_i) \in S} y_i \overline{X}_i^T (y_i [\overline{W} \cdot \overline{X}_i^T] - y_i^2)\end{aligned}$$

Setting $y_i^2 = 1$, we obtain the following:

$$\overline{W} \Leftarrow \overline{W}(1 - \alpha\lambda) + \alpha \sum_{(\overline{X}_i, y_i) \in S} y_i \overline{X}_i^T (1 - y_i [\overline{W} \cdot \overline{X}_i^T]) \tag{4.48}$$

This form of the update is more convenient because it is more closely related to updates of other classification models discussed later in this chapter. Examples of these models are the *support vector machine* and *logistic regression*. The loss function can also be converted to a more convenient representation for binary targets drawn from $\{-1, +1\}$.

Alternative Representation of Loss Function

The alternative form of the aforementioned updates can also be derived from an alternative form of the loss function. The loss function of (regularized) least-squares classification can be written as follows:

$$J = \frac{1}{2}\sum_{i=1}^{n}(y_i - \overline{W} \cdot \overline{X}_i^T)^2 + \frac{\lambda}{2}\|\overline{W}\|^2 \tag{4.49}$$

Using the fact that $y_i^2 = 1$ for binary targets, we can modify the objective function as follows:

$$\begin{aligned} J &= \frac{1}{2}\sum_{i=1}^{n} y_i^2(y_i - \overline{W} \cdot \overline{X}_i^T)^2 + \frac{\lambda}{2}\|\overline{W}\|^2 \\ &= \frac{1}{2}\sum_{i=1}^{n}(y_i^2 - y_i[\overline{W} \cdot \overline{X}_i^T])^2 + \frac{\lambda}{2}\|\overline{W}\|^2 \end{aligned}$$

Setting $y_i^2 = 1$, we obtain the following loss function:

$$J = \frac{1}{2}\sum_{i=1}^{n}(1 - y_i[\overline{W} \cdot \overline{X}_i^T])^2 + \frac{\lambda}{2}\|\overline{W}\|^2 \tag{4.50}$$

Differentiating this loss function directly leads to Equation 4.48. However, it is important to note that the loss function/updates of least-squares classification are identical to the loss function/updates of least-squares regression, even though one might use the binary nature of the targets in the former case in order to make them *look* superficially different.

The updates of least-squares classification are also referred to as Widrow-Hoff updates [132]. The rule was proposed in the context of neural network learning, and it was the second major neural learning algorithm proposed after the perceptron [109]. Interestingly, the neural models were proposed independently of the classical literature on least-squares regression; yet, the updates turn out to be identical.

Heuristic Initialization

A good way to perform heuristic initialization is to determine the mean $\overline{\mu}_0$ and $\overline{\mu}_1$ of the points belonging to the negative and positive classes, respectively. The difference between the two means is $\overline{w}_0 = \overline{\mu}_1^T - \overline{\mu}_0^T$ is a d-dimensional column vector, which satisfies $\overline{w}_0 \cdot \overline{\mu}_1^T \geq \overline{w}_0 \cdot \overline{\mu}_0^T$. The choice $\overline{W} = \overline{w}_0$ is a good starting point, because positive-class instances will have larger dot products with $\overline{w}_0$ than will negative-class instances (on the average). In many real applications, the classes are roughly separable with a linear hyperplane, and the normal hyperplane to the line joining the class centroids provides a good initial separator.

4.8.1.1 Why Least-Squares Classification Loss Needs Repair

The least-squares classification model has an important weakness, which is revealed when one examines its loss function:

$$J = \frac{1}{2}\sum_{i=1}^{n}(1 - y_i[\overline{W} \cdot \overline{X}_i^T])^2 + \frac{\lambda}{2}\|\overline{W}\|^2$$

Now consider a positive class instance for which $\overline{W} \cdot \overline{X}_i^T = 100$ is highly positive. This is obviously an desirable situation at least from a predictive point of view because the training instance is being classified on the correct side of the linear separator between the two classes in a positive way. However, the loss function in the training model treats this prediction as a large loss contribution of $(1 - y_i[\overline{W} \cdot \overline{X}_i^T])^2 = (1 - (1)(100))^2 = 99^2 = 9801$. Therefore, a large gradient descent update will be performed for a training instance that is located at a large distance from the hyperplane $\overline{W} \cdot \overline{X}^T = 0$ on the correct side. Such a situation is undesirable because it tends to confuse least-squares classification; the updates from these points on the correct side of the hyperplane $\overline{W} \cdot \overline{X}^T = 0$ tend to push the hyperplane in the same direction as some of the incorrectly classified points. In order to address this issue, many machine learning algorithms treat such points in a more nuanced way. These nuances will be discussed in the following sections.

4.8.2 The Support Vector Machine

As in the case of the least-squares classification model, we assume that we have n training pairs of the form $(\overline{X}_i, y_i)$ for $i \in \{1 \ldots n\}$. Each $\overline{X}_i$ is a d-dimensional row vector, and each $y_i \in \{-1, +1\}$ is the label. We would like to find a d-dimensional column vector $\overline{W}$ so that the sign of $\overline{W} \cdot \overline{X}_i^T$ yields the class label.

The support vector machine (SVM) treats *well-separated points* in the loss function in a more careful way by not penalizing them at all. What is a well separated point? Note that a point is correctly classified by the least-squares classification model when $y_i[\overline{W} \cdot \overline{X}_i^T] > 0$. In other words, y_i has the same sign as $\overline{W} \cdot \overline{X}_i^T$. Furthermore, the point is well-separated when $y_i[\overline{W} \cdot \overline{X}_i^T] > 1$. Therefore, the loss function of least-squares classification can be modified by setting the loss to 0, when this condition is satisfied. This can be achieved by modifying the least-squares loss to SVM loss as follows:

$$J = \frac{1}{2} \sum_{i=1}^{n} \max\left\{0, \left(1 - y_i[\overline{W} \cdot \overline{X}_i^T]\right)\right\}^2 + \frac{\lambda}{2}\|\overline{W}\|^2 \quad [L_2\text{-loss SVM}]$$

Note that the *only* difference from the least-squares classification model is the use of the maximization term in order to set the loss of well-separated points to 0. Once the vector $\overline{W}$ has been learned, the classification process for an unseen test instance is the same in the SVM as it is in the case of least-squares classification. For an unseen test instance $\overline{Z}$, the sign of $\overline{W} \cdot \overline{Z}^T$ yields the class label.

A more common form of the SVM loss is the *hinge-loss*. The hinge-loss is the L_1-version of the (squared) loss above:

$$J = \sum_{i=1}^{n} \max\{0, (1 - y_i[\overline{W} \cdot \overline{X}_i^T])\} + \frac{\lambda}{2}\|\overline{W}\|^2 \quad [\text{Hinge-loss SVM}] \tag{4.51}$$

Both forms of these objective functions can be shown to be convex.

Lemma 4.8.1 *Both the L_2-Loss SVM and the hinge loss are convex in the parameter vector $\overline{W}$. Furthermore, these functions are strictly convex when the regularization term is included.*

Proof: The proof of the above lemmas follow from the properties enumerated in Lemma 4.3.2. The point-specific hinge-loss is obtained by taking the maximum of two convex functions (one of which is linear and the other is a constant). Therefore, it is a convex

function as well. The L_2-loss SVM squares the nonnegative hinge loss. Since the square of a nonnegative convex function is convex (according to Lemma 4.3.2), it follows that the point-specific L_2-loss is convex. The sum of the point-specific losses (convex functions) is convex according to Lemma 4.3.2. Therefore, the unregularized loss is convex.

Regularized Loss: We have already shown earlier in Section 4.7.1 that the L_2-regularization term is strictly convex. Since the sum of a convex and a strictly convex function is strictly convex according to Lemma 4.3.6, both objective functions (including the regularization term) are strictly convex. ■

Therefore, one can find the *global* optimum of an SVM by using gradient descent.

4.8.2.1 Computing Gradients

The objective functions for the L_1-loss (hinge loss) and L_2-loss SVM are both in the form $J = \sum_i J_i + \Omega(\overline{W})$, where J_i is a point-specific loss and $\Omega(\overline{W}) = \lambda \|\overline{W}\|^2/2$ is the regularization term. The gradient of the latter term is $\lambda \overline{W}$. The main challenge is in computing the gradient of the point-specific loss J_i. Here, the key point is that the point-specific loss of both the L_1-loss (hinge loss) and L_2-loss can be expressed in the form of identity (v) of Table 4.2(a) for an appropriately chosen function $f(\cdot)$:

$$J_i = f_i(\overline{W} \cdot \overline{X}_i^T)$$

Here, the function $f_i(\cdot)$ is defined for the hinge-loss and L_2-loss SVMs as follows:

$$f_i(z) = \begin{cases} \max\{0, 1 - y_i z\} & \text{[Hinge Loss]} \\ \frac{1}{2}\max\{0, 1 - y_i z\}^2 & [L_2\text{-Loss}] \end{cases}$$

Therefore, according to Table 4.2(a) (also see Equation 4.29), the gradient of J_i with respect to $\overline{W}$ is the following:

$$\frac{\partial J_i}{\partial \overline{W}} = \overline{X}_i^T f_i'(\overline{W} \cdot \overline{X}_i^T) \tag{4.52}$$

The derivatives for the L_1-loss and the L_2-loss SVMs depend on the corresponding derivatives of $f_i(z)$, as they are defined in the two cases:

$$f_i'(z) = \begin{cases} -y_i I([1 - y_i z] > 0) & \text{[Hinge Loss]} \\ -y_i \max\{0, 1 - y_i z\} & [L_2\text{-Loss}] \end{cases}$$

Here, $I(\cdot)$ is an indicator function, which takes on the value of 1 when the condition inside it is true, and 0, otherwise. Therefore, by plugging in the value of $f'(z)$ in Equation 4.52, one obtains the following loss derivatives in the two cases:

$$\frac{\partial J_i}{\partial \overline{W}} = \begin{cases} -y_i \overline{X}_i^T I([1 - y_i(\overline{W} \cdot \overline{X}_i^T)] > 0) & \text{[Hinge Loss]} \\ -y_i \overline{X}_i^T \max\{0, 1 - y_i(\overline{W} \cdot \overline{X}_i^T)\} & [L_2\text{-Loss}] \end{cases}$$

These point-wise loss derivatives can be used to derive the stochastic gradient-descent updates.

4.8.2.2 Stochastic Gradient Descent

For the greatest generality, we will use mini-batch stochastic gradient descent in which a set S of training instances contains feature-label pairs of the form $(\overline{X}_i, y_i)$. For the hinge-loss SVM, we first determine the set $S^+ \subseteq S$ of training instances in which $y_i[\overline{W} \cdot \overline{X}_i^T] < 1$.

$$S^+ = \{(\overline{X}_i, y_i) : (\overline{X}_i, y_i) \in S, y_i[\overline{W} \cdot \overline{X}_i^T] < 1\} \tag{4.53}$$

The subset of instances in S^+ correspond to those for which the indicator function $I(\cdot)$ of the previous section takes on the value of 1. These instances are of two types; those corresponding to $y_i[\overline{W} \cdot \overline{X}_i^T] < 0$ are misclassified instances on the wrong side of the decision boundary, whereas the remaining instances corresponding to $y_i[\overline{W} \cdot \overline{X}_i^T] \in (0, 1)$ lie on the correct side of the decision boundary, but they are uncomfortably close to the decision boundary. Both these types of instances trigger updates in the SVM. In other words, the well-separated points do not play a role in the update. By using the gradient of the loss function, the updates in the L_1-loss SVM can be shown to be the following:

$$\overline{W} \Leftarrow \overline{W}(1 - \alpha\lambda) + \sum_{(\overline{X}_i, y_i) \in S^+} \alpha y_i \overline{X}_i^T \tag{4.54}$$

This algorithm is referred to as the primal support vector machine algorithm. The hinge-loss update seems somewhat different from the update for least-squares classification. The primary reason for this is that the least-squares classification model uses a squared loss function, whereas the hinge-loss is a piece-wise linear function. The similarity with the updates of least-squares classification becomes more obvious when one compares the updates of least-squares classification with those of the SVM with L_2-loss. The updates of the SVM with L_2-loss are as follows:

$$\overline{W} \Leftarrow \overline{W}(1 - \alpha\lambda) + \alpha \sum_{(\overline{X}_i, y_i) \in S} y_i \overline{X}_i^T (\max\{1 - y_i[\overline{W} \cdot \overline{X}_i^T], 0\}) \tag{4.55}$$

In this case, it is evident that the updates of the L_2-SVM are different from those of least-squares classification (cf. Equation 4.48) only in terms of the treatment of well-separated points; *identical updates are made for misclassified points and those near the decision boundary, whereas no updates are made for well-separated points on the correct side of the decision boundary.* This difference in the nature of the updates fully explains the difference between the L_2-SVM and least-squares classification. It is noteworthy that the loss function of the L_2-SVM was proposed [60] by Hinton much earlier than the Cortes and Vapnik [30] work on the hinge-loss SVM. Interestingly, Hinton proposed the L_2-loss as a way to repair the Widrow-Hoff loss (i.e., least-squares classification loss), which makes a lot of sense from an intuitive point of view. Hinton's work remained unnoticed by the community of researchers working on SVMs during the early years. However, the approach was eventually rediscovered in the recent focus on deep learning, where many of the early works were revisited.

4.8.3 Logistic Regression

We use the same notations as earlier sections by assuming that we have n training pairs of the form $(\overline{X}_i, y_i)$ for $i \in \{1 \ldots n\}$. Each $\overline{X}_i$ is a d-dimensional row vector, and each $y_i \in \{-1, +1\}$ is the label. We would like to find a d-dimensional column vector $\overline{W}$ so that the sign of $\overline{W} \cdot \overline{X}_i^T$ yields the class label of $\overline{X}_i$.

Logistic regression uses a loss function, which has a very similar shape to the hinge-loss SVM. However, the hinge-loss is piecewise linear, whereas logistic regression is a smooth loss function. Logistic regression has a probabilistic interpretation in terms of the log-likelihood loss of a data point. The loss function of logistic regression is formulated as follows:

$$J = \sum_{i=1}^{n} \underbrace{\log(1 + \exp(-y_i[\overline{W} \cdot \overline{X}_i^T]))}_{J_i} + \frac{\lambda}{2}\|\overline{W}\|^2 \quad \text{[Logistic Regression]} \tag{4.56}$$

All logarithms in this section are natural logarithms. When $\overline{W} \cdot \overline{X}_i^T$ is large in absolute magnitude and has the same sign as y_i, the point-specific loss J_i is close to $\log(1+\exp(-\infty)) = 0$. On the other hand, the loss is larger than $\log(1 + \exp(0)) = \log(2)$ when the signs of y_i and $\overline{W} \cdot \overline{X}_i^T$ disagree. For cases in which the signs disagree, the loss increases almost linearly with $\overline{W} \cdot \overline{X}_i^T$, as the magnitude of $\overline{W} \cdot \overline{X}_i^T$ becomes increasingly large. This is because of the following relationship:

$$\lim_{z \to -\infty} \frac{\log(1 + \exp(-z))}{-z} = \lim_{z \to -\infty} \frac{\exp(-z)}{1 + \exp(-z)} = \lim_{z \to -\infty} \frac{1}{1 + \exp(z)} = 1$$

The above limit is computed using *L'Hopital's rule*, which differentiates the numerator and denominator of a limit to evaluate it. Note that the hinge loss of an SVM is always $(1 - z)$ for $z = y_i \overline{W} \cdot \overline{X}_i^T < 1$. One can show that the logistic loss differs from the hinge loss by a constant offset of 1 for grossly misclassified instances:

Problem 4.8.1 *Show that* $\lim_{z \to -\infty} \underbrace{(1 - z)}_{SVM} - \underbrace{log(1 + exp(-z))}_{Logistic} = 1.$

Since constant offsets do not affect gradient descent, logistic loss and hinge loss treat grossly misclassified training instances in a similar way. However, unlike the hinge loss, all instances have non-zero logistic losses. Like SVMs, the loss function of logistic regression is convex:

Lemma 4.8.2 *The loss function of logistic regression is a convex function. Adding the regularization term makes the loss function strictly convex.*

Proof: This result can be shown by using the fact that the point-wise loss is of the form $\log[1+\exp(G(\overline{X}))]$, where $G(\overline{X}_i)$ is the linear function $G(\overline{X}_i) = -y_i(\overline{W} \cdot \overline{X}_i^T)$. Furthermore, the function $\log[1 + \exp(-z)]$ is convex (see Problem 4.3.4). Then, by using Lemma 4.3.2 on the composition of convex and linear functions, it is evident that each point-specific loss is convex. Adding all the point-specific losses also results in a convex function because of the first part of the same lemma. Furthermore, adding the regularization term makes the function strictly convex according to Lemma 4.3.6, because the regularization term is strictly convex. ■

It is, in fact, possible to show that logistic regression is strictly convex even without regularization. We leave the proof of this result as an exercise.

Problem 4.8.2 *Show that the loss function in logistic regression is strictly convex even without regularization.*

4.8.3.1 Computing Gradients

Since the logistic regression loss function is strictly convex, it means that one can reach a global optimum with stochastic gradient-descent methods. As in the case of SVMs, the objective function for logistic regression is in the form $J = \sum_i J_i + \Omega(\overline{W})$, where J_i is a point-specific loss and $\Omega(\overline{W}) = \lambda \|\overline{W}\|^2/2$ is the regularization term. The gradient of the regularization term is $\lambda \overline{W}$. We also need to compute the gradient of the point-specific loss J_i. The logistic loss can be expressed in the form of identity (v) of Table 4.2(a) for an appropriately chosen function $f(\cdot)$:

$$J_i = f_i(\overline{W} \cdot \overline{X}_i^T)$$

Here, the function $f_i(\cdot)$ is defined as follows for constant y_i:

$$f_i(z) = \log(1 + \exp(-y_i z))$$

Therefore, according to Table 4.2(a) (see also Equation 4.29), the gradient of J_i with respect to $\overline{W}$ is the following:

$$\frac{\partial J_i}{\partial \overline{W}} = \overline{X}_i^T f_i'(\overline{W} \cdot \overline{X}_i^T) \tag{4.57}$$

The corresponding derivative is as follows:

$$f_i'(z) = \frac{-y_i \exp(-y_i z)}{1 + \exp(-y_i z)} = \frac{-y_i}{1 + \exp(y_i z)}$$

Therefore, by plugging in the value of $f_i'(z)$ in Equation 4.57 after setting $z = \overline{W} \cdot \overline{X}_i^T$, one obtains the following loss derivative:

$$\frac{\partial J_i}{\partial \overline{W}} = \frac{-y_i \overline{X}_i^T}{(1 + \exp(y_i [\overline{W} \cdot \overline{X}_i^T]))}$$

These point-wise loss derivatives can be used to derive the stochastic gradient-descent updates.

4.8.3.2 Stochastic Gradient Descent

Given a mini-batch of S of feature-target pairs $(\overline{X}_i, y_i)$, one can define an objective function $J(S)$, which uses the loss of only the training instances in S. The regularization term remains unchanged, as one can simply re-scale the regularization parameter by $|S|/n$. It is relatively easy to compute the gradient $\nabla J(S)$ based on mini-batch S as follows:

$$\nabla J(S) = \lambda \overline{W} - \sum_{(\overline{X}_i, y_i) \in S} \frac{y_i \overline{X}_i^T}{(1 + \exp(y_i [\overline{W} \cdot \overline{X}_i^T]))} \tag{4.58}$$

Therefore, the mini-batch stochastic gradient-descent method can be implemented as follows:

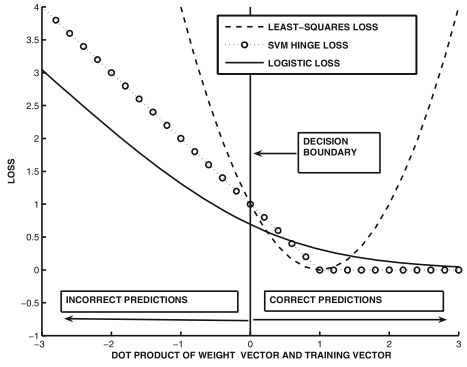
$$\overline{W} \Leftarrow \overline{W}(1 - \alpha\lambda) + \sum_{(\overline{X}_i, y_i) \in S} \frac{\alpha y_i \overline{X}_i^T}{(1 + \exp(y_i [\overline{W} \cdot \overline{X}_i^T]))} \tag{4.59}$$

Logistic regression makes similar updates as the hinge-loss SVM. The main difference is in terms of the treatment of well-separated points, where SVM does not make any updates and logistic regression makes (small) updates.

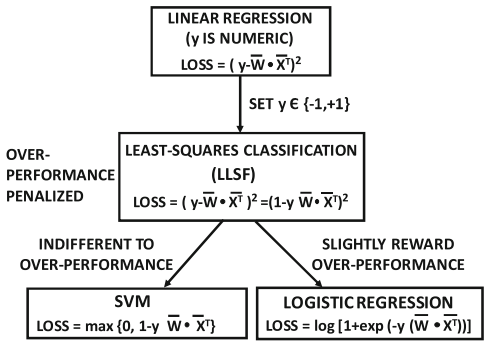
4.8.4 How Linear Regression Is a Parent Problem in Machine Learning

Many binary classification models use loss functions that are modifications of the least-squares regression loss function in order to handle binary target variables. The most extreme example of this inheritance is least-squares classification, where one directly uses the regression loss function by pretending that the labels from $\{-1, +1\}$ are numerical values. As discussed in Section 4.8.1.1, this direct inheritance of the regression loss function has undesirable consequences for binary data. In least-squares classification, the value of the loss first decreases as $\overline{W} \cdot \overline{X}^T$ increases as long as $\overline{W} \cdot \overline{X}^T \leq 1$; however, this loss increases for the same positive instance when $\overline{W} \cdot \overline{X}^T$ increases beyond 1. This is counter-intuitive behavior because one should not expect the loss to increase with increasingly correct classification of a point. After all, the sign of the predicted class label does not change with increasing positive values of $\overline{W} \cdot \overline{X}^T$. This situation is caused by the fact that least-squares classification is a blind application of linear regression to the classification problem, and it does not bother to make adjustments for the discrete nature of the class variable. In support-vector machines, increasing distance in the correct direction from the decision boundary beyond the point where $\overline{W} \cdot \overline{X}^T = 1$ is neither rewarded nor penalized, because the loss function is $\max\{1 - \overline{W} \cdot \overline{X}^T, 0\}$ (for positive class instances). This point is referred to as the *margin boundary* in support vector machines. In logistic regression, increasing distance of a training point $\overline{X}$ from the hyperplane $\overline{W} \cdot \overline{X}^T = 0$ on the correct side is slightly rewarded.

To show the differences among least-squares classification, SVM, and logistic regression, we have shown their loss at varying values of $\overline{W} \cdot \overline{X}^T$ of a **positive** training point $\overline{X}$ with label $y = +1$ [cf. Figure 4.9(a)]. Therefore, positive and increasing $\overline{W} \cdot \overline{X}^T$ is desirable for correct predictions. The loss functions of logistic regression and the support vector machine look strikingly similar, except that the former is a smooth function, and the SVM sharply bottoms at zero loss beyond $\overline{W} \cdot \overline{X}^T \geq 1$. This similarity in loss functions is also reflected



(a) Loss functions of optimization models



(b) Relationships among linear models

Figure 4.9: (a) The loss for a training instance $\overline{X}$ belonging to the **positive class** at varying values of $\overline{W} \cdot \overline{X}^T$. Logistic regression can be viewed as a smooth variant of SVM hinge loss. Least-squares classification is the only case in which the loss *increases* with increasingly correct classification in some regions. (b) All linear models in classification derive their motivation from the parent problem of linear regression

in the real-world experiences of machine learning practitioners who often find that the two models seem to provide similar results. The least-squares classification model provides the only loss function where increasing the magnitude of $\overline{W} \cdot \overline{X}^T$ increases the loss for correctly classified instances. The semantic relationships among different loss functions are illustrated in Figure 4.9(b). It is evident that all the binary classification models inherit the basic structure of their loss functions from least-squares regression (while making adjustments for the binary nature of the target variable).

These relationships among their loss functions are also reflected as relationships among their updates in gradient descent. The updates for all three models can be expressed in a unified way in terms of a model-specific *mistake function* $\delta(\overline{X}_i, y_i)$ for the training pair $(\overline{X}_i, y_i)$ at hand. In particular, it can be shown that the stochastic gradient-descent updates of all the above algorithms are of the following form:

$$\overline{W} \Leftarrow \overline{W}(1 - \alpha\lambda) + \alpha y_i[\delta(\overline{X}_i, y_i)]\overline{X}_i^T \tag{4.60}$$

The mistake function $\delta(\overline{X}_i, y_i)$ is $(y_i - \overline{W} \cdot \overline{X}_i^T)$ for least-squares regression and classification, an indicator variable for SVMs, and a probability value for logistic regression.

4.9 Optimization Models for the MultiClass Setting

In multi-class classification, the discrete labels are no longer binary. Rather, they are drawn from a set of k *unordered* possibilities, whose indices are $\{1, \ldots, k\}$. For example, the color of an object could be a label, and there is no ordering between the values of the targets. This lack of ordering of target attributes requires further algorithmic modifications.

Each training instance $(\overline{X}_i, c(i))$ contains a d-dimensional feature vector $\overline{X}_i$ (which is a row vector) and the index $c(i) \in \{1 \ldots k\}$ of its observed class. We would like to find k different column vectors $\overline{W}_1 \ldots \overline{W}_k$ simultaneously so that the value of $\overline{W}_{c(i)} \cdot \overline{X}_i^T$ is greater than $\overline{W}_r \cdot \overline{X}_i^T$ for each $r \neq c(i)$. In other words, the training instance $\overline{X}_i$ is predicted to the class r with the largest value of $\overline{W}_r \cdot \overline{X}_i^T$. After training, the test instances are predicted to the class with the largest dot product with the weight vector.

4.9.1 Weston-Watkins Support Vector Machine

For the ith training instance, $\overline{X}_i$, we would like $\overline{W}_{c(i)} \cdot \overline{X}_i^T - \overline{W}_j \cdot \overline{X}_i^T$ to be greater than 0 (for each $j \neq c(i)$). In keeping with the notion of margin in a support vector machine, we not only penalize incorrect classification, but also "barely correct" predictions. In other words, we would like to penalize cases in which $\overline{W}_{c(i)} \cdot \overline{X}_i^T - \overline{W}_j \cdot \overline{X}_i^T$ is less than some fixed positive value of the margin. This margin value can be set to 1, because using any other value a simply scales up the parameters by the same factor a. In other words, our "ideal" setting with zero loss is one in which the following is satisfied for each $j \neq c(i)$:

$$\overline{W}_{c(i)} \cdot \overline{X}_i^T - \overline{W}_j \cdot \overline{X}_i^T \geq 1 \tag{4.61}$$

Therefore, one can set up a loss value J_i for the ith training instance as follows:

$$J_i = \sum_{j:j \neq c(i)} \max(\overline{W}_j \cdot \overline{X}_i^T - \overline{W}_{c(i)} \cdot \overline{X}_i^T + 1, 0) \tag{4.62}$$

It is not difficult to see the similarity between this loss function and that of the binary SVM. The overall objective function can be computed by adding the losses over the different training instances, and also adding a regularization term $\Omega(\overline{W}_1 \ldots \overline{W}_k) = \lambda \sum_r \|\overline{W}_r\|^2/2$:

$$J = \sum_{i=1}^{n} \sum_{j:j \neq c(i)} \max(\overline{W}_j \cdot \overline{X}_i^T - \overline{W}_{c(i)} \cdot \overline{X}_i^T + 1, 0) + \frac{\lambda}{2} \sum_{r=1}^{k} \|\overline{W}_r\|^2$$

The fact that the Weston-Watkins loss function is convex has a proof that is very similar to the binary case. One needs to show that each additive term of J_i is convex in terms of the parameter vector; after all, this additive term is the composition of a linear and a maximization function. This can be used to show that J_i is convex as well. We leave this proof as an exercise for the reader:

Problem 4.9.1 *The Weston-Watkins loss function is convex in terms of its parameters.*

As in the case of the previous models, one can learn the weight vectors with the use of gradient descent.

4.9.1.1 Computing Gradients

The main point in computing gradients is the vector derivative of J_i with respect to $\overline{W}_r$. The above gradient is computed using the chain rule, while recognizing that J_i contains additive terms of the form $\max\{v_{ji}, 0\}$, where v_{ji} is defined as follows:

$$v_{ji} = \overline{W}_j \cdot \overline{X}_i^T - \overline{W}_{c(i)} \cdot \overline{X}_i^T + 1$$

Furthermore, the derivative of J_i can be written with respect to $\overline{W}_r$ by using the multivariate chain rule as follows:

$$\frac{\partial J_i}{\partial \overline{W}_r} = \sum_{j=1}^{k} \underbrace{\frac{\partial J_i}{\partial v_{ji}}}_{\delta(j, \overline{X}_i)} \frac{\partial v_{ji}}{\overline{W}_r} \tag{4.63}$$

The partial derivative of $J_i = \sum_r \max\{v_{ri}, 0\}$ with respect to v_{ji} is equal to the partial derivative of $\max\{v_{ji}, 0\}$ with respect to v_{ji}. The partial derivative of the function $\max\{v_{ji}, 0\}$ with respect to v_{ji} is 1 for positive v_{ji}, and 0, otherwise. We denote this value by $\delta(j, \overline{X}_i)$. In other words, the binary value $\delta(j, \overline{X}_i)$ is 1, when $\overline{W}_{c(i)} \cdot \overline{X}_i^T < \overline{W}_j \cdot \overline{X}_i^T + 1$, and therefore the correct class is not preferred with respect to class j with sufficient margin.

The right-hand side of Equation 4.63 requires us to compute the derivative of $v_{ji} = \overline{W}_j \cdot \overline{X}_i^T - \overline{W}_{c(i)} \cdot \overline{X}_i^T + 1$ with respect to $\overline{W}_r$. This is an easy derivative to compute because of its linearity, as long as we are careful to track which weight vectors $\overline{W}_r$ appear with positive signs in v_{ji}. In the case when $r \neq c(i)$ (separator for wrong class), the derivative of v_{ji} with respect to $\overline{W}_r$ is $\overline{X}_i^T$ when $j = r$, and 0, otherwise. In the case when $r = c(i)$, the derivative is $-\overline{X}_i^T$ when $j \neq r$, and 0, otherwise. On substituting these values, one obtains the gradient of J_i with respect to $\overline{W}_r$ as follows:

$$\frac{\partial J_i}{\partial \overline{W}_r} = \begin{cases} \delta(r, \overline{X}_i)\overline{X}_i^T & r \neq c(i) \\ -\sum_{j \neq r} \delta(j, \overline{X}_i)\overline{X}_i^T & r = c(i) \end{cases}$$

One can obtain the gradient of J with respect to $\overline{W}_r$ by summing up the contributions of the different J_i and the regularization component of $\lambda \overline{W}_r$. Therefore, the updates for stochastic gradient descent are as follows:

$$\begin{aligned}\overline{W}_r &\Leftarrow \overline{W}_r(1-\alpha\lambda) - \alpha \frac{\partial J_i}{\partial \overline{W}_r} \quad \forall r \in \{1 \ldots k\} \\ &= \overline{W}_r(1-\alpha\lambda) - \alpha \begin{cases} \delta(r, \overline{X}_i)\overline{X}_i^T & r \neq c(i) \\ -\sum_{j \neq r} \delta(j, \overline{X}_i)\overline{X}_i^T & r = c(i) \end{cases} \quad \forall r \in \{1 \ldots k\}\end{aligned}$$

An important special case is one in which there are only two classes. In such a case, it can be shown that the resulting updates of the separator belonging to the positive class will be identical to those in the hinge-loss SVM. Furthermore, the relationship $\overline{W}_1 = -\overline{W}_2$ will always be maintained, assuming that the parameters are initialized in this way. This is because the update to each separator will be the negative of the update to the other separator. We leave the proof of this result as a practice exercise.

Problem 4.9.2 *Show that the Weston-Watkins SVM defaults to the binary hinge-loss SVM in the special case of two classes.*

One observation from the relationship $\overline{W}_1 = -\overline{W}_2$ in the binary case is that there is a slight redundancy in the number of parameters of the multiclass SVM. This is because we really need $(k-1)$ separators in order to model k classes, and one separator is redundant. However, since the update of the kth separator is always exactly defined by the updates of the other $(k-1)$ separators, this redundancy does not make a difference.

Problem 4.9.3 *Propose a natural L_2-loss function for the multclass SVM. Derive the gradient and the details of stochastic gradient descent in this case.*

4.9.2 Multinomial Logistic Regression

Multinomial logistic regression is a generalization of logistic regression to multiple classes. As in the case of the Weston-Watkins SVM, each training instance $(\overline{X}_i, c(i))$ contains a d-dimensional feature vector $\overline{X}_i$ (which is a row vector) and the index $c(i) \in \{1 \ldots k\}$ of its observed class. Furthermore, similar to the Weston-Watkins SVM, k different separators are learned whose parameter vectors are $\overline{W}_1 \ldots \overline{W}_k$. The prediction rule for test instances is also the same as the Weston-Watkins SVM, since the class j with the largest dot product $\overline{W}_j \cdot \overline{Z}^T$ is predicted as the class of test instance $\overline{Z}$. Multinomial logistic regression models the *probability* of a point belonging to the rth class. The probability of training point $\overline{X}_i$ belonging to class r is given by applying the *softmax function* to $\overline{W}_1 \cdot \overline{X}_i^T \ldots \overline{W}_k \cdot \overline{X}_i^T$:

$$P(r|\overline{X}_i) = \frac{\exp(\overline{W}_r \cdot \overline{X}_i^T)}{\sum_{j=1}^{k} \exp(\overline{W}_j \cdot \overline{X}_i^T)} \tag{4.64}$$

It is easy to verify that the probability of $\overline{X}_i$ belonging to the rth class increases exponentially with increasing dot product between $\overline{W}_r$ and $\overline{X}_i^T$.

The goal in learning $\overline{W}_1 \ldots \overline{W}_k$ is to ensure that the aforementioned probability is high for the class $c(i)$ for (each) instance $\overline{X}_i$. This is achieved by using the *cross-entropy loss*,

which is the negative logarithm of the probability of the instance $\overline{X}_i$ belonging to the correct class $c(i)$:

$$J = -\sum_{i=1}^{n} \underbrace{\log[P(c(i)|\overline{X}_i)]}_{J_i} + \frac{\lambda}{2}\sum_{r=1}^{k} \|\overline{W}_r\|^2$$

It is relatively easy to show that each $J_i = -\log[P(c(i)|\overline{X}_i)]$ is convex using an approach similar to the case of binary logistic regression.

4.9.2.1 Computing Gradients

We would like the determine the gradient of J with respect to each $\overline{W}_r$. We can decompose this gradient into the sum of the gradients of $J_i = -\log[P(c(i)|\overline{X}_i)]$ (along with the gradient of the regularization term). We denote this quantity by $\frac{\partial J_i}{\partial \overline{W}_r}$. Let v_{ji} denote the quantity $\overline{W}_j \cdot \overline{X}_i^T$. Then, the value of $\frac{\partial J_i}{\partial \overline{W}_r}$ is computed using the chain rule as follows:

$$\frac{\partial J_i}{\partial \overline{W}_r} = \sum_j \left(\frac{\partial J_i}{\partial v_{ji}}\right) \frac{\partial v_{ji}}{\partial \overline{W}_r} = \frac{\partial J_i}{\partial v_{ri}} \underbrace{\frac{\partial v_{ri}}{\partial \overline{W}_r}}_{\overline{X}_i^T} = \overline{X}_i^T \frac{\partial J_i}{\partial v_{ri}} \tag{4.65}$$

In the above simplification, we used the fact that v_{ji} has a zero gradient with respect to $\overline{W}_r$ for $j \neq r$, and therefore all terms in the summation except for the case of $j = r$ drop out to 0. We still need to compute the partial derivative of J_i with respect to v_{ri}. First, we express J_i directly as a function of $v_{1i}, v_{2i}, \ldots, v_{ki}$ as follows:

$$J_i = -\log[P(c(i)|\overline{X}_i)] = -\overline{W}_{c(i)} \cdot \overline{X}_i^T + \log[\sum_{j=1}^{k} \exp(\overline{W}_j \cdot \overline{X}_i^T)] \quad \text{[Using Equation 4.64]}$$

$$= -v_{c(i),i} + \log[\sum_{j=1}^{k} \exp(v_{ji})]$$

Therefore, we can compute the partial derivative of J_i with respect to v_{ri} as follows:

$$\frac{\partial J_i}{\partial v_{ri}} = \begin{cases} -\left(1 - \frac{\exp(v_{ri})}{\sum_{j=1}^{k} \exp(v_{ji})}\right) & \text{if } r = c(i) \\ \left(\frac{\exp(v_{ri})}{\sum_{j=1}^{k} \exp(v_{ji})}\right) & \text{if } r \neq c(i) \end{cases}$$

$$= \begin{cases} -(1 - P(r|\overline{X}_i)) & \text{if } r = c(i) \\ P(r|\overline{X}_i) & \text{if } r \neq c(i) \end{cases}$$

By substituting the value of the partial derivative $\frac{\partial J_i}{\partial v_{ri}}$ in Equation 4.65, we obtain the following:

$$\frac{\partial J_i}{\partial \overline{W}_r} = \begin{cases} -\overline{X}_i^T (1 - P(r|\overline{X}_i)) & \text{if } r = c(i) \\ \overline{X}_i^T P(r|\overline{X}_i) & \text{if } r \neq c(i) \end{cases} \tag{4.66}$$

4.9.2.2 Stochastic Gradient Descent

One can then use this point-specific gradient to compute the stochastic gradient descent updates:

$$\overline{W}_r \Leftarrow \overline{W}_r(1-\alpha\lambda) + \alpha \begin{cases} \overline{X}_i^T(1 - P(r|\overline{X}_i)) & \text{if } r = c(i) \\ -\overline{X}_i^T P(r|\overline{X}_i) & \text{if } r \neq c(i) \end{cases} \quad \forall r \in \{1 \ldots k\} \tag{4.67}$$

The probabilities in the above update can be substituted using Equation 4.64. It is noteworthy that the updates use the *probabilities of mistakes* in order to change each separator. In comparison, methods like least-squares regression use the *magnitudes of mistakes* in the updates. This difference is natural, because the softmax method is a probabilistic model. The above stochastic gradient descent is proposed for a mini-batch size of 1. We leave the derivation for a mini-batch S as an exercise for the reader.

Problem 4.9.4 *The text provides the derivation of stochastic gradient descent in multinomial logistic regression for a mini-batch size of 1. Provide a derivation of the update of each separator $\overline{W}_r$ for a mini-batch S containing pairs of the form $(\overline{X}, c)$ as follows:*

$$\overline{W}_r \Leftarrow \overline{W}_r(1-\alpha\lambda) + \alpha \sum_{(\overline{X},c)\in S, r=c} \overline{X}^T \cdot (1 - P(r|\overline{X})) - \alpha \sum_{(\overline{X},c)\in S, r\neq c} \overline{X}^T \cdot P(r|\overline{X}) \tag{4.68}$$

Just as the Weston-Watkins SVM defaults to the hinge-loss SVM for the two-class case, multinomial logistic regression defaults to logistic regression in the special case of two classes. We leave the proof of this result as an exercise.

Problem 4.9.5 *Show that multinomial logistic regression defaults to binary logistic regression in the special case of two classes.*

4.10 Coordinate Descent

Coordinate descent is a method that optimizes the objective function one variable at a time. Therefore, if we have an objective function $J(\overline{w})$, which is a function of d-dimensional vector variables, we can try to optimize a single variable w_i from the vector $\overline{w}$, while holding all the other parameters fixed. This corresponds to the following optimization problem:

$$\overline{w} = \text{argmin}_{[w_i \text{ varies only}]} \; J(\overline{w}) \qquad [\text{All parameters except } w_i \text{ are fixed}]$$

Note that this is a single-variable optimization problem, which is usually much simpler to solve. In some cases, one might need to use line-search to determine w_i, when a closed form of the solution is not available. If one cycles through all the variables, and no improvement occurs, convergence has occurred. In the event that the optimized function is convex and differentiable in minimization form, the solution at convergence will be the optimal one. For non-convex functions, optimality is certainly not guaranteed, as the system can get stuck at a local minimum. Even for functions that are convex but non-differentiable, it is possible for coordinate descent to reach a suboptimal solution. An important point about coordinate descent is that it implicitly uses more than first-order gradient information; after all, it finds an optimal solution with respect to the variable it is optimizing. As a result, convergence can sometimes be faster with coordinate descent, as compared to stochastic

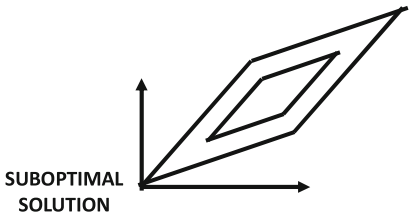


Figure 4.10: The contour plot of a non-differentiable function is shown. The center of the parallelogram-like contour plot is the optimum. Note that the axis-parallel moves can only worsen the objective function from acute-angled positions

gradient descent. Another important point about coordinate descent is that convergence is usually guaranteed, even if the resulting solution is a local optimum.

There are two main problems with coordinate descent. First, it is inherently sequential in nature. The approach optimizes one variable at a time, and therefore it would need to have optimized with respect to one variable in order to perform the next optimization step. Therefore, the parallelization of coordinate descent is always a challenge. Second, it can get stuck at suboptimal points (local minima). Even though the convergence to a local minimum is guaranteed, the use of a single variable can sometimes be myopic. This type of problem could occur even for convex functions, if the function is not differentiable. For example, consider the following function:

$$f(x, y) = |x + y| + 2|x - y| \tag{4.69}$$

This objective function is convex but not differentiable. The optimal point of this function is $(0, 0)$. However, if coordinate descent reaches the point $(1, 1)$, it will cycle through both variables without improving the solution. The problem is that *no path exists to the optimal solution using axis-parallel directions*. Such a situation can occur with non-differentiable functions having pointed contour plots; if one ends up at one of the corners of the contour plot, there might not be a suitable axis-parallel direction of movement in order to improve the objective function. An example of such a scenario is illustrated in Figure 4.10. Such a situation can never arise in a differentiable function, where at least one axis-parallel direction will always improve the objective function.

A natural question that arises is to characterize the conditions under which coordinate descent is well behaved in non-differentiable function optimization. One observation is that even though the function $f(x, y)$ of Equation 4.69 is convex, its additive components are not separable in terms of the individual variables. In general, a sufficient condition for coordinate descent to reach a global optimum solution is that *the additive components of the non-differentiable portion of the multivariate function need to be expressed in terms of individual variables, and each of them must be convex*. We summarize a general version of the above result:

Lemma 4.10.1 *Consider a multivariate function $F(\overline{w})$ that can be expressed in the following form:*

$$F(\overline{w}) = G(\overline{w}) + \sum_{i=1}^{d} H_i(w_i)$$

The function $G(\overline{w})$ is a convex and differentiable function, whereas each $H_i(w_i)$ is a convex, univariate function of w_i, which might be non-differentiable. Then, coordinate descent will converge to a global optimum of the function $F(\overline{w})$.

An example of a non-differentiable function $H_i(w_i)$, which is also convex, is $H_i(w_i) = |w_i|$. This function is used for L_1-regularization. In fact, we will discuss the use of coordinate descent for L_1-regularized regression in Section 5.8.1.2 of Chapter 5.

The issue of additive separability is important, and it is sometimes helpful to perform a variable transformation, so that the non-differentiable part is additively separable. For example, consider a generalization of the objective function of Equation 4.69:

$$f(x, y) = g(x, y) + |x + y| + 2|x - y| \tag{4.70}$$

Assume that $g(x, y)$ is differentiable. Now, we make the following variable transformations $u = x + y$ and $v = x - y$. Then, one can rewrite the objective function after the variable transformation as $f([u + v]/2, [u - v]/2)$. In other words, we always substitute $[u + v]/2$ everywhere for x and $[u - v]/2$ everywhere for y to obtain the following:

$$F(u, v) = g([u + v]/2, [u - v]/2) + |u| + 2|v| \tag{4.71}$$

Each of the non-differentiable components is a convex function. Now, one can perform coordinate descent with respect to u and v without any problem. The main point of this trick is that the variable transformation changes the directions of movement, so that a path to the optimum solution exists.

Interestingly, even though non-differentiable functions cause problems for coordinate descent, such functions (and even discrete optimization problems) are often better solved by coordinate descent than gradient descent. This is because coordinate descent often enables the decomposition of a complex problem into smaller subproblems. As a specific example of this decomposition, we will show how the well-known k-means algorithm is an example of coordinate descent, when applied to a potentially difficult *mixed integer program* (cf. Section 4.10.3).

4.10.1 Linear Regression with Coordinate Descent

Consider an $n \times d$ data matrix D (with rows containing training instances), an n-dimensional column vector $\overline{y}$ of response variables, and a d-dimensional column vector $\overline{W} = [w_1 \ldots w_d]^T$ of parameters. We revisit the linear-regression objective function of Equation 4.31 as follows:

$$J = \frac{1}{2}\|D\overline{W} - \overline{y}\|^2 \tag{4.72}$$

The corresponding gradient with respect to *all* variables is used in straightforward gradient-descent methods (cf. Equation 4.33):

$$\nabla J = D^T(D\overline{W} - \overline{y}) \tag{4.73}$$

Coordinate descent optimizes the objective with respect to only a *single* variable at a time. In order to optimize with respect to w_i, we need to pick out the ith component of ∇J and set it to zero. Let $\overline{d}_i$ be the ith column of D. Furthermore, let $\overline{r}$ denote the n-dimensional residual vector $\overline{y} - D\overline{W}$. Then, we obtain the following condition:

$$\begin{aligned}
&\overline{d}_i^T(D\overline{W} - \overline{y}) = 0 \\
&\overline{d}_i^T(\overline{r}) = 0 \\
&\overline{d}_i^T\overline{r} + w_i\overline{d}_i^T\overline{d}_i = w_i\overline{d}_i^T\overline{d}_i
\end{aligned}$$

Note that the left-hand side is free of w_i because the two terms involving w_i cancel each other out. This is because the term $\overline{d}_i^T \overline{r}$ contributes $-w_i \overline{d}_i^T \overline{d}_i$, which cancels with $w_i \overline{d}_i^T \overline{d}_i$. Because of the fact that one of the sides does not depend on w_i, we obtain an update that yields the optimal value of w_i in a *single* iteration:

$$\overline{w}_i \Leftarrow \overline{w}_i + \frac{\overline{d}_i^T \overline{r}}{\|\overline{d}_i\|^2} \tag{4.74}$$

In the above update, we have used the fact that $\overline{d}_i^T \overline{d}_i$ is the same as the squared norm of $\overline{d}_i$. It is common to standardize each column of the data matrix to zero mean and unit variance. In such a case, the value of $\|\overline{d}_i\|^2$ will be 1, and the update further simplifies to the following:

$$\overline{w}_i \Leftarrow \overline{w}_i + \overline{d}_i^T \overline{r} \tag{4.75}$$

This update is extremely efficient. One full cycle of coordinate descent through all the variables requires asymptotically similar time as one full cycle of stochastic gradient descent through all the points. However, the number of cycles required by coordinate descent tends to be smaller than that in least-squares regression. Therefore, the coordinate-descent approach is more efficient. One can also derive a form of coordinate descent for regularized least-squares regression. We leave this problem as a practice exercise.

Problem 4.10.1 *Show that if Tikhonov regularization is used with parameter λ on least-squares regression, then the update of Equation 4.74 needs to be modified to the following:*

$$w_i \Leftarrow \frac{w_i \|\overline{d}_i\|^2 + \overline{d}_i^T \overline{r}}{\|\overline{d}_i\|^2 + \lambda}$$

The simplification of optimization subproblems that are inherent in solving for one variable at a time (while keeping others fixed) is very significant in coordinate descent.

4.10.2 Block Coordinate Descent

Block coordinate descent generalizes coordinate descent by optimizing a *block* of variables at a time, rather than a single variable. Although each step in block coordinate descent is more expensive, fewer steps are required. An example of block coordinate descent is the *alternating least-squares method*, which is often used in matrix factorization (cf Section 8.3.2.3 of Chapter 8). Block coordinate descent is often used in multi-convex problems where the objective function is non-convex, but each block of variables can be used to create a convex subproblem. Alternatively, each block admits to easy optimization, even when some of the variables are discrete. It is sometimes also easy to handle constrained optimization problems with coordinate descent, because the constraints tend to simplify themselves, when one is considering only a few carefully chosen variables. A specific example of this type of setting is the k-means algorithm.

4.10.3 K-Means as Block Coordinate Descent

The k-means algorithm is a good example of how choosing specific blocks of variables carefully allows good alternating minimization over different blocks of variables. One often views k-means as a simple heuristic method, although the reality is that it is fundamentally rooted in important ideas from coordinate descent.

It is assumed that there are a total of n data points denoted by the d-dimensional row vectors $\overline{X}_1 \ldots \overline{X}_n$. The k-means algorithms creates k *prototypes*, which are denoted by $\overline{z}_1 \ldots \overline{z}_k$, so that the sum of squared distances of the data points from their nearest prototypes is as small as possible. Let y_{ij} be a 0-1 indicator of whether point i gets assigned to cluster j. Each point gets assigned to only a single cluster, and therefore we have $\sum_j y_{ij} = 1$. One can therefore, formulate the k-means problem as a *mixed integer program* over the *real-valued* d-dimensional prototype row vectors $\overline{z}_1 \ldots \overline{z}_k$ and the matrix $Y = [y_{ij}]_{n \times k}$ of *discrete* assignment variables:

$$\text{Minimize} \underbrace{\sum_{j=1}^{k} \sum_{i=1}^{n} y_{ij} \|\overline{X}_i - \overline{z}_j\|^2}_{O_j}$$

$$\text{subject to:}$$

$$\sum_{j=1}^{k} y_{ij} = 1$$

$$y_{ij} \in \{0, 1\}$$

This is a mixed integer program, and such optimization problems are known to be very hard to solve in general. However, in this case, carefully choosing the blocks of variables is essential. Choosing the blocks of variables carefully also trivializes the underlying constraints. In this particular case, the variables are divided into two blocks corresponding to the $k \times d$ prototype variables in the vectors $\overline{z}_1 \ldots \overline{z}_k$ and the $n \times k$ assignment variables $Y = [y_{ij}]$. We alternately minimize over these two blocks of variables, because it provides the best possible decomposition of the problem into smaller subproblems. Note that if the prototype variables are fixed, the resulting assignment problem becomes trivial and one assigns each point to the nearest prototype. On the other hand, if the cluster assignments are fixed, then the objective function can be decomposed into separate objective functions over different clusters. The portion of the objective function O_j contributed by the jth cluster is shown by an underbrace in the optimization formulation above. For each cluster, the relevant optimal solution $\overline{z}_j$ is the mean of the points assigned to that cluster. This result can be shown by setting the gradient of the objective function O_j with respect to each $\overline{z}_j$ to 0:

$$\frac{\partial O_j}{\partial z_j} = 2 \sum_{i=1}^{n} y_{ij} (\overline{X}_i - \overline{z}_j) = 0 \quad \forall j \in \{1 \ldots k\} \tag{4.76}$$

The points that do not belong to cluster j drop out in the above condition because $y_{ij} = 0$ for such points. As a result, $\overline{z}_j$ is simply the mean of the points in its cluster. Therefore, we need to alternative assign points to their closest prototypes, and set the prototypes to the centroids of the clusters defined by the assignment; these are exactly the steps of the well-known k-means algorithm. The centroid computation is a continuous optimization step, whereas cluster assignment is a discrete optimization step (which is greatly simplified by the decomposition approach of coordinate descent).

4.11 Summary

This chapter introduces the basic optimization models in machine learning. We discussed the conditions for optimality, as well as the cases in which a global optimum is guaranteed. Optimization problems in machine learning often have objective functions which can be